MXNet Documentation

Release 0.0.8

pluskid

Nov 09, 2017

Tutorials

MXNet.jl is Julia package of dmlc/mxnet. MXNet.jl brings flexible and efficient GPU computing and state-of-art deep learning to Julia. Some highlight of features include:

- Efficient tensor/matrix computation across multiple devices, including multiple CPUs, GPUs and distributed server nodes.
- Flexible symbolic manipulation to composite and construct state-of-the-art deep learning models.

For more details, see documentation below. Please also checkout the examples directory.

CHAPTER 1

Digit Recognition on MNIST

In this tutorial, we will work through examples of training a simple multi-layer perceptron and then a convolutional neural network (the LeNet architecture) on the MNIST handwritten digit dataset. The code for this tutorial could be found in examples/mnist.

1.1 Simple 3-layer MLP

This is a tiny 3-layer MLP that could be easily trained on CPU. The script starts with

```
using MXNet
```

to load the `MXNet` module. Then we are ready to define the network architecture via the *symbolic API*. We start with a placeholder `data` symbol,

```
data = mx.Variable(:data)
```

and then cascading fully-connected layers and activation functions:

```
fc1  = mx.FullyConnected(data = data, name=:fc1, num_hidden=128)
act1 = mx.Activation(data = fc1, name=:relu1, act_type=:relu)
fc2  = mx.FullyConnected(data = act1, name=:fc2, num_hidden=64)
act2 = mx.Activation(data = fc2, name=:relu2, act_type=:relu)
fc3  = mx.FullyConnected(data = act2, name=:fc3, num_hidden=10)
```

Note each composition we take the previous symbol as the *data* argument, forming a feedforward chain. The architecture looks like

```
Input --> 128 units (ReLU) --> 64 units (ReLU) --> 10 units
```

where the last 10 units correspond to the 10 output classes (digits 0,. . . ,9). We then add a final `SoftmaxOutput` operation to turn the 10-dimensional prediction to proper probability values for the 10 classes:

```
mlp  = mx.SoftmaxOutput(data = fc3, name=:softmax)
```

As we can see, the MLP is just a chain of layers. For this case, we can also use the mx.chain macro. The same architecture above can be defined as

```
mlp = @mx.chain mx.Variable(:data)             =>
  mx.FullyConnected(name=:fc1, num_hidden=128) =>
  mx.Activation(name=:relu1, act_type=:relu)   =>
  mx.FullyConnected(name=:fc2, num_hidden=64)  =>
  mx.Activation(name=:relu2, act_type=:relu)   =>
  mx.FullyConnected(name=:fc3, num_hidden=10)  =>
  mx.SoftmaxOutput(name=:softmax)
```

After defining the architecture, we are ready to load the MNIST data. MXNet.jl provide built-in data providers for the MNIST dataset, which could automatically download the dataset into Pkg.dir("MXNet")/data/mnist if necessary. We wrap the code to construct the data provider into mnist-data.jl so that it could be shared by both the MLP example and the LeNet ConvNets example.

```
batch_size = 100
include("mnist-data.jl")
train_provider, eval_provider = get_mnist_providers(batch_size)
```

If you need to write your own data providers for customized data format, please refer to AbstractDataProvider.

Given the architecture and data, we can instantiate an *model* to do the actual training. mx.FeedForward is the built-in model that is suitable for most feed-forward architectures. When constructing the model, we also specify the *context* on which the computation should be carried out. Because this is a really tiny MLP, we will just run on a single CPU device.

```
model = mx.FeedForward(mlp, context=mx.cpu())
```

You can use a mx.gpu() or if a list of devices (e.g. [mx.gpu(0), mx.gpu(1)]) is provided, data-parallelization will be used automatically. But for this tiny example, using a GPU device might not help.

The last thing we need to specify is the optimization algorithm (a.k.a. *optimizer*) to use. We use the basic SGD with a fixed learning rate 0.1 and momentum 0.9:

```
optimizer = mx.SGD(lr=0.1, momentum=0.9, weight_decay=0.00001)
```

Now we can do the training. Here the n_epoch parameter specifies that we want to train for 20 epochs. We also supply a eval_data to monitor validation accuracy on the validation set.

```
mx.fit(model, optimizer, train_provider, n_epoch=20, eval_data=eval_provider)
```

Here is a sample output

```
INFO: Start training on [CPU0]
INFO: Initializing parameters...
INFO: Creating KVStore...
INFO: == Epoch 001 ==========
INFO: ## Training summary
INFO:       :accuracy = 0.7554
INFO:            time = 1.3165 seconds
INFO: ## Validation summary
INFO:       :accuracy = 0.9502
...
INFO: == Epoch 020 ==========
INFO: ## Training summary
```

```
INFO:       :accuracy = 0.9949
INFO:            time = 0.9287 seconds
INFO: ## Validation summary
INFO:       :accuracy = 0.9775
```

1.2 Convolutional Neural Networks

In the second example, we show a slightly more complicated architecture that involves convolution and pooling. This architecture for the MNIST is usually called the *[LeNet]*. The first part of the architecture is listed below:

```
# input
data = mx.Variable(:data)

# first conv
conv1 = @mx.chain mx.Convolution(data=data, kernel=(5,5), num_filter=20)  =>
                  mx.Activation(act_type=:tanh) =>
                  mx.Pooling(pool_type=:max, kernel=(2,2), stride=(2,2))

# second conv
conv2 = @mx.chain mx.Convolution(data=conv1, kernel=(5,5), num_filter=50) =>
                  mx.Activation(act_type=:tanh) =>
                  mx.Pooling(pool_type=:max, kernel=(2,2), stride=(2,2))
```

We basically defined two convolution modules. Each convolution module is actually a chain of `Convolution`, `tanh` activation and then max `Pooling` operations.

Each sample in the MNIST dataset is a 28x28 single-channel grayscale image. In the tensor format used by `NDArray`, a batch of 100 samples is a tensor of shape `(28,28,1,100)`. The convolution and pooling operates in the spatial axis, so `kernel=(5,5)` indicate a square region of 5-width and 5-height. The rest of the architecture follows as:

```
# first fully-connected
fc1   = @mx.chain mx.Flatten(data=conv2) =>
                  mx.FullyConnected(num_hidden=500) =>
                  mx.Activation(act_type=:tanh)

# second fully-connected
fc2   = mx.FullyConnected(data=fc1, num_hidden=10)

# softmax loss
lenet = mx.Softmax(data=fc2, name=:softmax)
```

Note a fully-connected operator expects the input to be a matrix. However, the results from spatial convolution and pooling are 4D tensors. So we explicitly used a `Flatten` operator to flat the tensor, before connecting it to the `FullyConnected` operator.

The rest of the network is the same as the previous MLP example. As before, we can now load the MNIST dataset:

```
batch_size = 100
include("mnist-data.jl")
train_provider, eval_provider = get_mnist_providers(batch_size; flat=false)
```

Note we specified `flat=false` to tell the data provider to provide 4D tensors instead of 2D matrices because the convolution operators needs correct spatial shape information. We then construct a feedforward model on GPU, and train it.

```
#--------------------------------------------------------------------------------
# fit model
model = mx.FeedForward(lenet, context=mx.gpu())

# optimizer
optimizer = mx.SGD(lr=0.05, momentum=0.9, weight_decay=0.00001)

# fit parameters
mx.fit(model, optimizer, train_provider, n_epoch=20, eval_data=eval_provider)
```

And here is a sample of running outputs:

```
INFO: == Epoch 001 ==========
INFO: ## Training summary
INFO:       :accuracy = 0.6750
INFO:            time = 4.9814 seconds
INFO: ## Validation summary
INFO:       :accuracy = 0.9712
...
INFO: == Epoch 020 ==========
INFO: ## Training summary
INFO:       :accuracy = 1.0000
INFO:            time = 4.0086 seconds
INFO: ## Validation summary
INFO:       :accuracy = 0.9915
```

1.3 Predicting with a trained model

Predicting with a trained model is very simple. By calling `mx.predict` with the model and a data provider, we get the model output as a Julia Array:

```
probs = mx.predict(model, eval_provider)
```

The following code shows a stupid way of getting all the labels from the data provider, and compute the prediction accuracy manually:

```
# collect all labels from eval data
labels = Array[]
for batch in eval_provider
  push!(labels, copy(mx.get_label(batch)))
end
labels = cat(1, labels...)

# Now we use compute the accuracy
correct = 0
for i = 1:length(labels)
  # labels are 0...9
  if indmax(probs[:,i]) == labels[i]+1
    correct += 1
  end
end
println(mx.format("Accuracy on eval set: {1:.2f}%", 100correct/length(labels)))
```

Alternatively, when the dataset is huge, one can provide a callback to `mx.predict`, then the callback function will be invoked with the outputs of each mini-batch. The callback could, for example, write the data to disk for future inspection. In this case, no value is returned from `mx.predict`. See also `predict()`.

CHAPTER 2

Generating Random Sentence with LSTM RNN

This tutorial shows how to train a LSTM (Long short-term memory) RNN (recurrent neural network) to perform character-level sequence training and prediction. The original model, usually called `char-rnn` is described in Andrej Karpathy's blog, with a reference implementation in Torch available here.

Because MXNet.jl does not have a specialized model for recurrent neural networks yet, the example shown here is an implementation of LSTM by using the default `FeedForward` model via explicitly unfolding over time. We will be using fixed-length input sequence for training. The code is adapted from the char-rnn example for MXNet's Python binding, which demonstrates how to use low-level *symbolic APIs* to build customized neural network models directly.

The most important code snippets of this example is shown and explained here. To see and run the complete code, please refer to the examples/char-lstm directory. You will need to install Iterators.jl and StatsBase.jl to run this example.

2.1 LSTM Cells

Christopher Olah has a great blog post about LSTM with beautiful and clear illustrations. So we will not repeat the definition and explanation of what an LSTM cell is here. Basically, an LSTM cell takes input `x`, as well as previous states (including `c` and `h`), and produce the next states. We define a helper type to bundle the two state variables together:

```
immutable LSTMState
  c :: mx.SymbolicNode
  h :: mx.SymbolicNode
end
```

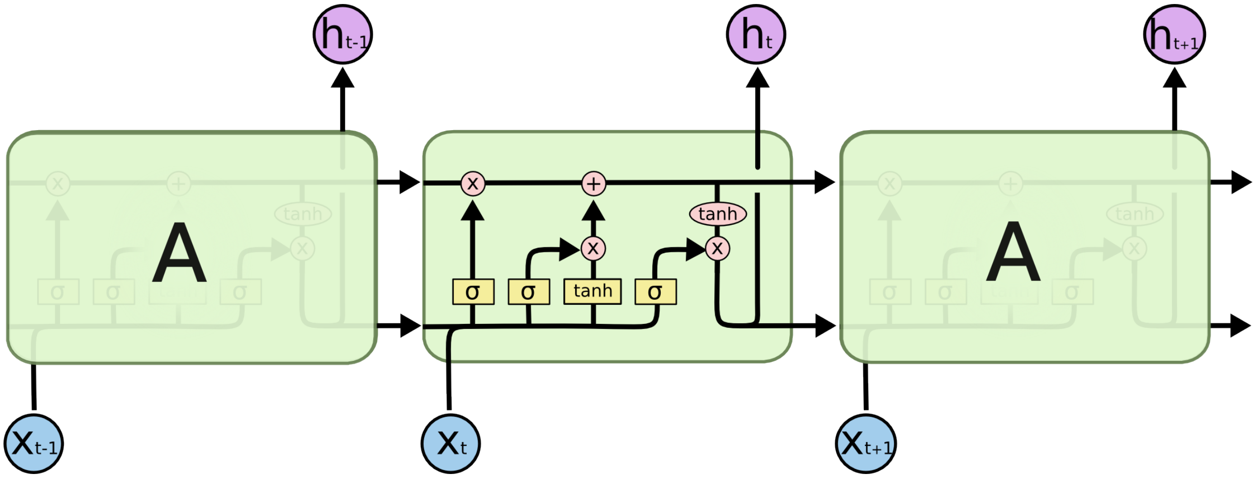
Because LSTM weights are shared at every time when we do explicit unfolding, so we also define a helper type to hold all the weights (and bias) for an LSTM cell for convenience.

```
immutable LSTMParam
  i2h_W :: mx.SymbolicNode
  h2h_W :: mx.SymbolicNode
  i2h_b :: mx.SymbolicNode
  h2h_b :: mx.SymbolicNode
end
```

Note all the variables are of type `SymbolicNode`. We will construct the LSTM network as a symbolic computation graph, which is then instantiated with `NDArray` for actual computation.

```
function lstm_cell(data::mx.SymbolicNode, prev_state::LSTMState, param::LSTMParam;
                   num_hidden::Int=512, dropout::Real=0, name::Symbol=gensym())

  if dropout > 0
    data = mx.Dropout(data, p=dropout)
  end

  i2h = mx.FullyConnected(data=data, weight=param.i2h_W, bias=param.i2h_b,
                          num_hidden=4num_hidden, name=symbol(name, "_i2h"))
  h2h = mx.FullyConnected(data=prev_state.h, weight=param.h2h_W, bias=param.h2h_b,
                          num_hidden=4num_hidden, name=symbol(name, "_h2h"))

  gates = mx.SliceChannel(i2h + h2h, num_outputs=4, name=symbol(name, "_gates"))

  in_gate     = mx.Activation(gates[1], act_type=:sigmoid)
  in_trans    = mx.Activation(gates[2], act_type=:tanh)
  forget_gate = mx.Activation(gates[3], act_type=:sigmoid)
  out_gate    = mx.Activation(gates[4], act_type=:sigmoid)

  next_c = (forget_gate .* prev_state.c) + (in_gate .* in_trans)
  next_h = out_gate .* mx.Activation(next_c, act_type=:tanh)

  return LSTMState(next_c, next_h)
end
```

The following figure is stolen (permission requested) from Christopher Olah's blog, which illustrate exactly what the code snippet above is doing.



In particular, instead of defining the four gates independently, we do the computation together and then use `SliceChannel` to split them into four outputs. The computation of gates are all done with the symbolic API. The return value is a LSTM state containing the output of a LSTM cell.

2.2 Unfolding LSTM

Using the LSTM cell defined above, we are now ready to define a function to unfold a LSTM network with L layers and T time steps. The first part of the function is just defining all the symbolic variables for the shared weights and states.

The `embed_W` is the weights used for character embedding — i.e. mapping the one-hot encoded characters into real vectors. The `pred_W` and `pred_b` are weights and bias for the final prediction at each time step.

Then we define the weights for each LSTM cell. Note there is one cell for each layer, and it will be replicated (unrolled) over time. The states are, however, *not* shared over time. Instead, here we define the initial states here at the beginning of a sequence, and we will update them with the output states at each time step as we explicitly unroll the LSTM.

```
function LSTM(n_layer::Int, seq_len::Int, dim_hidden::Int, dim_embed::Int, n_
↪class::Int;
              dropout::Real=0, name::Symbol=gensym(), output_states::Bool=false)

  # placeholder nodes for all parameters
  embed_W = mx.Variable(symbol(name, "_embed_weight"))
  pred_W  = mx.Variable(symbol(name, "_pred_weight"))
  pred_b  = mx.Variable(symbol(name, "_pred_bias"))

  layer_param_states = map(1:n_layer) do i
    param = LSTMParam(mx.Variable(symbol(name, "_l$(i)_i2h_weight")),
                      mx.Variable(symbol(name, "_l$(i)_h2h_weight")),
                      mx.Variable(symbol(name, "_l$(i)_i2h_bias")),
                      mx.Variable(symbol(name, "_l$(i)_h2h_bias")))
    state = LSTMState(mx.Variable(symbol(name, "_l$(i)_init_c")),
                      mx.Variable(symbol(name, "_l$(i)_init_h")))
    (param, state)
  end
  #...
```

Unrolling over time is a straightforward procedure of stacking the embedding layer, and then LSTM cells, on top of which the prediction layer. During unrolling, we update the states and collect all the outputs. Note each time step takes data and label as inputs. If the LSTM is named as `:ptb`, the data and label at step `t` will be named `:ptb_data_$t` and `:ptb_label_$t`. Late on when we prepare the data, we will define the data provider to match those names.

```
  # now unroll over time
  outputs = mx.SymbolicNode[]
  for t = 1:seq_len
    data   = mx.Variable(symbol(name, "_data_$t"))
    label  = mx.Variable(symbol(name, "_label_$t"))
    hidden = mx.FullyConnected(data=data, weight=embed_W, num_hidden=dim_embed,
                               no_bias=true, name=symbol(name, "_embed_$t"))

    # stack LSTM cells
    for i = 1:n_layer
      l_param, l_state = layer_param_states[i]
      dp = i == 1 ? 0 : dropout # don't do dropout for data
      next_state = lstm_cell(hidden, l_state, l_param, num_hidden=dim_hidden,␣
↪dropout=dp,
                             name=symbol(name, "_lstm_$t"))
      hidden = next_state.h
      layer_param_states[i] = (l_param, next_state)
    end

    # prediction / decoder
    if dropout > 0
      hidden = mx.Dropout(hidden, p=dropout)
    end
    pred = mx.FullyConnected(data=hidden, weight=pred_W, bias=pred_b, num_hidden=n_
↪class,
                             name=symbol(name, "_pred_$t"))
    smax = mx.SoftmaxOutput(pred, label, name=symbol(name, "_softmax_$t"))
    push!(outputs, smax)
  end
  #...
```

Note at each time step, the prediction is connected to a `SoftmaxOutput` operator, which could back propagate when corresponding labels are provided. The states are then connected to the next time step, which allows back propagate through time. However, at the end of the sequence, the final states are not connected to anything. This dangling outputs is problematic, so we explicitly connect each of them to a `BlockGrad` operator, which simply back propagates 0-gradient and closes the computation graph.

In the end, we just group all the prediction outputs at each time step as a single `SymbolicNode` and return. Optionally we will also group the final states, this is used when we use the trained LSTM to sample sentences.

```
  # append block-gradient nodes to the final states
  for i = 1:n_layer
    l_param, l_state = layer_param_states[i]
    final_state = LSTMState(mx.BlockGrad(l_state.c, name=symbol(name, "_l$(i)_last_c
↪")),
                            mx.BlockGrad(l_state.h, name=symbol(name, "_l$(i)_last_h
↪")))
    layer_param_states[i] = (l_param, final_state)
  end

  # now group all outputs together
  if output_states
    outputs = outputs  [x[2].c for x in layer_param_states]
                        [x[2].h for x in layer_param_states]
  end
  return mx.Group(outputs...)
end
```

2.3 Data Provider for Text Sequences

Now we need to construct a data provider that takes a text file, divide the text into mini-batches of fixed-length character-sequences, and provide them as one-hot encoded vectors.

Note the is no fancy feature extraction at all. Each character is simply encoded as a one-hot vector: a 0-1 vector of the size given by the vocabulary. Here we just construct the vocabulary by collecting all the unique characters in the training text – there are not too many of them (including punctuations and whitespace) for English text. Each input character is then encoded as a vector of 0s on all coordinates, and 1 on the coordinate corresponding to that character. The character-to-coordinate mapping is giving by the vocabulary.

The text sequence data provider implement the *data provider API*. We define the `CharSeqProvider` as below:

```
type CharSeqProvider <: mx.AbstractDataProvider
  text       :: AbstractString
  batch_size :: Int
  seq_len    :: Int
  vocab      :: Dict{Char,Int}
  prefix     :: Symbol
  n_layer    :: Int
  dim_hidden :: Int
end
```

The provided data and labels follow the naming convention of inputs used when unrolling the LSTM. Note in the code below, apart from `$name_data_$t` and `$name_label_$t`, we also provides the initial `c` and `h` states for each layer. This is because we are using the high-level `FeedForward` API, which has no idea about time and states. So we will feed the initial states for each sequence from the data provider. Since the initial states is always zero, we just need to always provide constant zero blobs.

```
function mx.provide_data(p :: CharSeqProvider)
  [(symbol(p.prefix, "_data_$t"), (length(p.vocab), p.batch_size)) for t = 1:p.seq_
↪len]
  [(symbol(p.prefix, "_l$(l)_init_c"), (p.dim_hidden, p.batch_size)) for l=1:p.n_
↪layer]
  [(symbol(p.prefix, "_l$(l)_init_h"), (p.dim_hidden, p.batch_size)) for l=1:p.n_
↪layer]
end
function mx.provide_label(p :: CharSeqProvider)
  [(symbol(p.prefix, "_label_$t"), (p.batch_size,)) for t = 1:p.seq_len]
end
```

Next we implement the `AbstractDataProvider.eachbatch()` interface for the provider. We start by defining the data and label arrays, and the `DataBatch` object we will provide in each iteration.

```
function mx.eachbatch(p :: CharSeqProvider)
  data_all  = [mx.zeros(shape) for (name, shape) in mx.provide_data(p)]
  label_all = [mx.zeros(shape) for (name, shape) in mx.provide_label(p)]

  data_jl = [copy(x) for x in data_all]
  label_jl= [copy(x) for x in label_all]

  batch = mx.DataBatch(data_all, label_all, p.batch_size)
  #...
```

The actual data providing iteration is implemented as a Julia **coroutine**. In this way, we can write the data loading logic as a simple coherent `for` loop, and do not need to implement the interface functions like `Base.start()`, `Base.next()`, etc.

Basically, we partition the text into batches, each batch containing several contiguous text sequences. Note at each time step, the LSTM is trained to predict the next character, so the label is the same as the data, but shifted ahead by one index.

```
  #...
  function _text_iter()
    text = p.text

    n_batch = floor(Int, length(text) / p.batch_size / p.seq_len)
    text = text[1:n_batch*p.batch_size*p.seq_len] # discard tailing
    idx_all = 1:length(text)

    for idx_batch in partition(idx_all, p.batch_size*p.seq_len)
      for i = 1:p.seq_len
        data_jl[i][:] = 0
        label_jl[i][:] = 0
      end

      for (i, idx_seq) in enumerate(partition(idx_batch, p.seq_len))
        for (j, idx) in enumerate(idx_seq)
          c_this = text[idx]
          c_next = idx == length(text) ? UNKNOWN_CHAR : text[idx+1]
          data_jl[j][char_idx(vocab,c_this),i] = 1
          label_jl[j][i] = char_idx(vocab,c_next)-1
        end
      end

      for i = 1:p.seq_len
        copy!(data_all[i], data_jl[i])
        copy!(label_all[i], label_jl[i])
      end

      produce(batch)
    end
  end

  return Task(_text_iter)
end
```

2.4 Training the LSTM

Now we have implemented all the supporting infrastructures for our char-lstm. To train the model, we just follow the standard high-level API. Firstly, we construct a LSTM symbolic architecture:

```
# define LSTM
lstm = LSTM(LSTM_N_LAYER, SEQ_LENGTH, DIM_HIDDEN, DIM_EMBED,
            n_class, dropout=DROPOUT, name=NAME)
```

Note all the parameters are defined in examples/char-lstm/config.jl. Now we load the text file and define the data provider. The data `input.txt` we used in this example is a tiny Shakespeare dataset. But you can try with other text files.

```
# load data
text_all  = readall(INPUT_FILE)
len_train = round(Int, length(text_all)*DATA_TR_RATIO)
text_tr   = text_all[1:len_train]
text_val  = text_all[len_train+1:end]

data_tr   = CharSeqProvider(text_tr, BATCH_SIZE, SEQ_LENGTH, vocab, NAME,
                            LSTM_N_LAYER, DIM_HIDDEN)
data_val  = CharSeqProvider(text_val, BATCH_SIZE, SEQ_LENGTH, vocab, NAME,
                            LSTM_N_LAYER, DIM_HIDDEN)
```

The last step is to construct a model, an optimizer and fit the mode to the data. We are using the `ADAM` optimizer *[Adam]* in this example.

```
model = mx.FeedForward(lstm, context=context)
optimizer = mx.ADAM(lr=BASE_LR, weight_decay=WEIGHT_DECAY, grad_clip=CLIP_GRADIENT)

mx.fit(model, optimizer, data_tr, eval_data=data_val, n_epoch=N_EPOCH,
       initializer=mx.UniformInitializer(0.1),
       callbacks=[mx.speedometer(), mx.do_checkpoint(CKPOINT_PREFIX)], eval_
 ↪metric=NLL())
```

Note we are also using a customized NLL evaluation metric, which calculate the negative log-likelihood during training. Here is an output sample at the end of the training process.

```
...
INFO: Speed: 357.72 samples/sec
INFO: == Epoch 020 ==========
INFO: ## Training summary
INFO:                NLL = 1.4672
INFO:         perplexity = 4.3373
INFO:               time = 87.2631 seconds
INFO: ## Validation summary
INFO:                NLL = 1.6374
INFO:         perplexity = 5.1418
INFO: Saved checkpoint to 'char-lstm/checkpoints/ptb-0020.params'
INFO: Speed: 368.74 samples/sec
INFO: Speed: 361.04 samples/sec
INFO: Speed: 360.02 samples/sec
INFO: Speed: 362.34 samples/sec
INFO: Speed: 360.80 samples/sec
INFO: Speed: 362.77 samples/sec
INFO: Speed: 357.18 samples/sec
INFO: Speed: 355.30 samples/sec
INFO: Speed: 362.33 samples/sec
INFO: Speed: 359.23 samples/sec
INFO: Speed: 358.09 samples/sec
INFO: Speed: 356.89 samples/sec
INFO: Speed: 371.91 samples/sec
INFO: Speed: 372.24 samples/sec
INFO: Speed: 356.59 samples/sec
INFO: Speed: 356.64 samples/sec
INFO: Speed: 360.24 samples/sec
INFO: Speed: 360.32 samples/sec
INFO: Speed: 362.38 samples/sec
INFO: == Epoch 021 ==========
INFO: ## Training summary
INFO:                NLL = 1.4655
INFO:         perplexity = 4.3297
INFO:               time = 86.9243 seconds
INFO: ## Validation summary
INFO:                NLL = 1.6366
INFO:         perplexity = 5.1378
INFO: Saved checkpoint to 'examples/char-lstm/checkpoints/ptb-0021.params'
```

2.5 Sampling Random Sentences

After training the LSTM, we can now sample random sentences from the trained model. The sampler works in the following way:

- Starting from some fixed character, take a for example, and feed it as input to the LSTM.
- The LSTM will produce an output distribution over the vocabulary and a state in the first time step. We sample a character from the output distribution, fix it as the second character.

- In the next time step, we feed the previously sampled character as input and continue running the LSTM by also taking the previous states (instead of the 0 initial states).
- Continue running until we sampled enough characters.

Note we are running with mini-batches, so several sentences could be sampled simultaneously. Here are some sampled outputs from a network I trained for around half an hour on the Shakespeare dataset. Note all the line-breaks, punctuations and upper-lower case letters are produced by the sampler itself. I did not do any post-processing.

```
## Sample 1
all have sir,
Away will fill'd in His time, I'll keep her, do not madam, if they here? Some more ha?

## Sample 2
am.

CLAUDIO:
Hone here, let her, the remedge, and I know not slept a likely, thou some soully free?

## Sample 3
arrel which noble thing
The exchnachsureding worns: I ne'er drunken Biancas, fairer, than the lawfu?

## Sample 4
augh assalu, you'ld tell me corn;
Farew. First, for me of a loved. Has thereat I knock you presents?

## Sample 5
ame the first answer.

MARIZARINIO:
Door of Angelo as her lord, shrield liken Here fellow the fool ?

## Sample 6
ad well.

CLAUDIO:
Soon him a fellows here; for her fine edge in a bogms' lord's wife.

LUCENTIO:
I?

## Sample 7
adrezilian measure.

LUCENTIO:
So, help'd you hath nes have a than dream's corn, beautio, I perchas?

## Sample 8
as eatter me;
The girlly: and no other conciolation!

BISTRUMIO:
I have be rest girl. O, that I a h?

## Sample 9
and is intend you sort:
What held her all 'clama's for maffice. Some servant.' what I say me the cu?
```

```
## Sample 10
an thoughts will said in our pleasue,
Not scanin on him that you live; believaries she.

ISABELLLLL?
```

See Andrej Karpathy's blog post on more examples and links including Linux source codes, Algebraic Geometry Theorems, and even cooking recipes. The code for sampling can be found in examples/char-lstm/sampler.jl.

2.6 Visualizing the LSTM

Finally, you could visualize the LSTM by calling `to_graphviz()` on the constructed LSTM symbolic architecture. We only show an example of 1-layer and 2-time-step LSTM below. The automatic layout produced by GraphViz is definitely much less clear than Christopher Olah's illustrations, but could otherwise be very useful for debugging. As we can see, the LSTM unfolded over time is just a (very) deep neural network. The complete code for producing this visualization can be found in examples/char-lstm/visualize.jl.

CHAPTER 3

Installation Guide

3.1 Automatic Installation

To install MXNet.jl, simply type

```
Pkg.add("MXNet")
```

in the Julia REPL. Or to use the latest git version of MXNet.jl, use the following command instead

```
Pkg.checkout("MXNet")
```

MXNet.jl is built on top of libmxnet. Upon installation, Julia will try to automatically download and build libmxnet.

The libmxnet source is downloaded to `Pkg.dir("MXNet")/deps/src/mxnet`. The automatic build is using default configurations, with OpenCV, CUDA disabled. If the compilation failed due to unresolved dependency, or if you want to customize the build, it is recommended to compile and install libmxnet manually. Please see below for more details.

3.2 Manual Compilation

It is possible to compile libmxnet separately and point MXNet.jl to a the existing library in case automatic compilation fails due to unresolved dependencies in an un-standard environment; Or when one want to work with a seperate, maybe customized libmxnet.

To build libmxnet, please refer to the installation guide of libmxnet. After successfully installing libmxnet, set the `MXNET_HOME` environment variable to the location of libmxnet. In other words, the compiled `libmxnet.so` should be found in `$MXNET_HOME/lib`.

Note: The constant `MXNET_HOME` is pre-compiled in MXNet.jl package cache. If you updated the environment variable after installing MXNet.jl, make sure to update the pre-compilation cache by `Base.compilecache("MXNet")`.

When the `MXNET_HOME` environment variable is detected and the corresponding `libmxnet.so` could be loaded successfully, MXNet.jl will skip automatic building during installation and use the specified libmxnet instead.

Basically, MXNet.jl will search `libmxnet.so` or `libmxnet.dll` in the following paths (and in that order):

- `$MXNET_HOME/lib`: customized libmxnet builds
- `Pkg.dir("MXNet")/deps/usr/lib`: automatic builds
- Any system wide library search path

CHAPTER 4

Overview

4.1 MXNet.jl Namespace

Most the functions and types in MXNet.jl are organized in a flat namespace. Because many some functions are conflicting with existing names in the Julia Base module, we wrap them all in a `mx` module. The convention of accessing the MXNet.jl interface is the to use the `mx.` prefix explicitly:

```
using MXNet

x = mx.zeros(2,3)              # MXNet NDArray
y = zeros(eltype(x), size(x))  # Julia Array
copy!(y, x)                    # Overloaded function in Julia Base
z = mx.ones(size(x), mx.gpu()) # MXNet NDArray on GPU
mx.copy!(z, y)                 # Same as copy!(z, y)
```

Note functions like `size`, `copy!` that is extensively overloaded for various types works out of the box. But functions like `zeros` and `ones` will be ambiguous, so we always use the `mx.` prefix. If you prefer, the `mx.` prefix can be used explicitly for all MXNet.jl functions, including `size` and `copy!` as shown in the last line.

4.2 Low Level Interface

4.2.1 NDArrays

`NDArray` is the basic building blocks of the actual computations in MXNet. It is like a Julia `Array` object, with some important differences listed here:

- The actual data could live on different `Context` (e.g. GPUs). For some contexts, iterating into the elements one by one is very slow, thus indexing into `NDArray` is not supported in general. The easiest way to inspect the contents of an `NDArray` is to use the `copy` function to copy the contents as a Julia `Array`.
- Operations on `NDArray` (including basic arithmetics and neural network related operators) are executed in parallel with automatic dependency tracking to ensure correctness.

- There is no generics in `NDArray`, the `eltype` is always `mx.MX_float`. Because for applications in machine learning, single precision floating point numbers are typical a best choice balancing between precision, speed and portability. Also since libmxnet is designed to support multiple languages as front-ends, it is much simpler to implement with a fixed data type.

While most of the computation is hidden in libmxnet by operators corresponding to various neural network layers. Getting familiar with the `NDArray` API is useful for implementing `Optimizer` or customized operators in Julia directly.

The followings are common ways to create `NDArray` objects:

- `mx.empty(shape[, context])`: create on uninitialized array of a given shape on a specific device. For example, ``mx.empty(2,3)`, `mx.((2,3), mx.gpu(2)).``
- `mx.zeros(shape[, context])` and `mx.ones(shape[, context])`: similar to the Julia's built-in `zeros` and `ones`.
- `mx.copy(jl_arr, context)`: copy the contents of a Julia `Array` to a specific device.

Most of the convenient functions like `size`, `length`, `ndims`, `eltype` on array objects should work out-of-the-box. Although indexing is not supported, it is possible to take *slices*:

```
a = mx.ones(2,3)
b = mx.slice(a, 1:2)
b[:] = 2
println(copy(a))
# =>
# Float32[2.0 2.0 1.0
#         2.0 2.0 1.0]
```

A slice is a sub-region sharing the same memory with the original `NDArray` object. A slice is always a contiguous piece of memory, so only slicing on the *last* dimension is supported. The example above also shows a way to set the contents of an `NDArray`.

```
a = mx.empty(2,3)
a[:] = 0.5              # set all elements to a scalar
a[:] = rand(size(a))    # set contents with a Julia Array
copy!(a, rand(size(a))) # set value by copying a Julia Array
b = mx.empty(size(a))
b[:] = a                # copying and assignment between NDArrays
```

Note due to the intrinsic design of the Julia language, a normal assignment

```
a = b
```

does **not** mean copying the contents of `b` to `a`. Instead, it just make the variable `a` pointing to a new object, which is `b`. Similarly, inplace arithmetics does not work as expected:

```
a = mx.ones(2)
r = a           # keep a reference to a
b = mx.ones(2)
a += b          # translates to a = a + b
println(copy(a))
# => Float32[2.0f0,2.0f0]
println(copy(r))
# => Float32[1.0f0,1.0f0]
```

As we can see, `a` has expected value, but instead of inplace updating, a new `NDArray` is created and `a` is set to point to this new object. If we look at `r`, which still reference to the old `a`, its content has not changed. There is currently no way in Julia to overload the operators like += to get customized behavior.

Instead, you will need to write `a[:] = a+b`, or if you want *real* inplace += operation, MXNet.jl provides a simple macro `@mx.inplace`:

```
@mx.inplace a += b
macroexpand(:(@mx.inplace a += b))
# => :(MXNet.mx.add_to!(a,b))
```

As we can see, it translate the += operator to an explicit `add_to!` function call, which invokes into libmxnet to add the contents of `b` into `a` directly. For example, the following is the update rule in the SGD `Optimizer` (both `grad` and `weight` are `NDArray` objects):

```
@inplace weight += -lr * (grad_scale * grad + self.weight_decay * weight)
```

Note there is no much magic in `mx.inplace`: it only does a shallow translation. In the SGD update rule example above, the computation like scaling the gradient by `grad_scale` and adding the weight decay all create temporary `NDArray` objects. To mitigate this issue, libmxnet has a customized memory allocator designed specifically to handle this kind of situations. The following snippet does a simple benchmark on allocating temp `NDArray` vs. pre-allocating:

```
using Benchmark
using MXNet

N_REP = 1000
SHAPE = (128, 64)
CTX   = mx.cpu()
LR    = 0.1

function inplace_op()
  weight = mx.zeros(SHAPE, CTX)
  grad   = mx.ones(SHAPE, CTX)

  # pre-allocate temp objects
  grad_lr = mx.empty(SHAPE, CTX)

  for i = 1:N_REP
    copy!(grad_lr, grad)
    @mx.inplace grad_lr .*= LR
    @mx.inplace weight -= grad_lr
  end
  return weight
end

function normal_op()
  weight = mx.zeros(SHAPE, CTX)
  grad   = mx.ones(SHAPE, CTX)

  for i = 1:N_REP
    weight[:] -= LR * grad
  end
  return weight
end

# make sure the results are the same
@assert(maximum(abs(copy(normal_op() - inplace_op()))) < 1e-6)

println(compare([inplace_op, normal_op], 100))
```

The comparison on my laptop shows that `normal_op` while allocating a lot of temp `NDArray` in the loop (the performance gets worse when increasing N_REP), is only about twice slower than the pre-allocated one.

Row	Function	Average	Relative	Replications
1	“inplace_op”	0.0074854	1.0	100
2	“normal_op”	0.0174202	2.32723	100

So it will usually not be a big problem unless you are at the bottleneck of the computation.

4.2.2 Distributed Key-value Store

The type KVStore and related methods are used for data sharing across different devices or machines. It provides a simple and efficient integer - NDArray key-value storage system that each device can pull or push.

The following example shows how to create a local KVStore, initialize a value and then pull it back.

```
kv    = mx.KVStore(:local)
shape = (2,3)
key   = 3

mx.init!(kv, key, mx.ones(shape)*2)
a = mx.empty(shape)
mx.pull!(kv, key, a) # pull value into a
println(copy(a))
# =>
# Float32[2.0 2.0 2.0
#        2.0 2.0 2.0]
```

4.3 Intermediate Level Interface

4.3.1 Symbols and Composition

The way we build deep learning models in MXNet.jl is to use the powerful symbolic composition system. It is like Theano, except that we avoided long expression compiliation time by providing *larger* neural network related building blocks to guarantee computation performance. See also this note for the design and trade-off of the MXNet symbolic composition system.

The basic type is mx.Symbol. The following is a trivial example of composing two symbols with the + operation.

```
A = mx.Variable(:A)
B = mx.Variable(:B)
C = A + B
```

We get a new *symbol* by composing existing *symbols* by some *operations*. A hierarchical architecture of a deep neural network could be realized by recursive composition. For example, the following code snippet shows a simple 2-layer MLP construction, using a hidden layer of 128 units and a ReLU activation function.

```
net = mx.Variable(:data)
net = mx.FullyConnected(data=net, name=:fc1, num_hidden=128)
net = mx.Activation(data=net, name=:relu1, act_type=:relu)
net = mx.FullyConnected(data=net, name=:fc2, num_hidden=64)
net = mx.Softmax(data=net, name=:out)
```

Each time we take the previous symbol, and compose with an operation. Unlike the simple + example above, the *operations* here are "bigger" ones, that correspond to common computation layers in deep neural networks.

Each of those operation takes one or more input symbols for composition, with optional hyper-parameters (e.g. `num_hidden`, `act_type`) to further customize the composition results.

When applying those operations, we can also specify a `name` for the result symbol. This is convenient if we want to refer to this symbol later on. If not supplied, a name will be automatically generated.

Each symbol takes some arguments. For example, in the + case above, to compute the value of `C`, we will need to know the values of the two inputs `A` and `B`. For neural networks, the arguments are primarily two categories: *inputs* and *parameters*. *inputs* are data and labels for the networks, while *parameters* are typically trainable *weights*, *bias*, *filters*.

When composing symbols, their arguments accumulates. We can list all the arguments by

```
julia> mx.list_arguments(net)
6-element Array{Symbol,1}:
 :data          # Input data, name from the first data variable
 :fc1_weight    # Weights of the fully connected layer named :fc1
 :fc1_bias      # Bias of the layer :fc1
 :fc2_weight    # Weights of the layer :fc2
 :fc2_bias      # Bias of the layer :fc2
 :out_label     # Input label, required by the softmax layer named :out
```

Note the names of the arguments are generated according to the provided name for each layer. We can also specify those names explicitly:

```
net = mx.Variable(:data)
w   = mx.Variable(:myweight)
net = mx.FullyConnected(data=data, weight=w, name=:fc1, num_hidden=128)
mx.list_arguments(net)
# =>
# 3-element Array{Symbol,1}:
#  :data
#  :myweight
#  :fc1_bias
```

The simple fact is that a `Variable` is just a placeholder `mx.Symbol`. In composition, we can use arbitrary symbols for arguments. For example:

```
net  = mx.Variable(:data)
net  = mx.FullyConnected(data=net, name=:fc1, num_hidden=128)
net2 = mx.Variable(:data2)
net2 = mx.FullyConnected(data=net2, name=:net2, num_hidden=128)
mx.list_arguments(net2)
# =>
# 3-element Array{Symbol,1}:
#  :data2
#  :net2_weight
#  :net2_bias
composed_net = net2(data2=net, name=:composed)
mx.list_arguments(composed_net)
# =>
# 5-element Array{Symbol,1}:
#  :data
#  :fc1_weight
#  :fc1_bias
#  :net2_weight
#   :net2_bias
```

Note we use a composed symbol, net as the argument data2 for net2 to get a new symbol, which we named :composed. It also shows that a symbol itself is a call-able object, which can be invoked to fill in missing arguments and get more complicated symbol compositions.

4.3.2 Shape Inference

Given enough information, the shapes of all arguments in a composed symbol could be inferred automatically. For example, given the input shape, and some hyper-parameters like num_hidden, the shapes for the weights and bias in a neural network could be inferred.

```
net = mx.Variable(:data)
net = mx.FullyConnected(data=net, name=:fc1, num_hidden=10)
arg_shapes, out_shapes, aux_shapes = mx.infer_shape(net, data=(10, 64))
```

The returned shapes corresponds to arguments with the same order as returned by mx.list_arguments. The out_shapes are shapes for outputs, and aux_shapes can be safely ignored for now.

```
for (n,s) in zip(mx.list_arguments(net), arg_shapes)
  println("$n => $s")
end
# =>
# data => (10,64)
# fc1_weight => (10,10)
# fc1_bias => (10,)
for (n,s) in zip(mx.list_outputs(net), out_shapes)
  println("$n => $s")
end
# =>
# fc1_output => (10,64)
```

4.3.3 Binding and Executing

In order to execute the computation graph specified a composed symbol, we will *bind* the free variables to concrete values, specified as mx.NDArray. This will create an mx.Executor on a given mx.Context. A context describes the computation devices (CPUs, GPUs, etc.) and an executor will carry out the computation (forward/backward) specified in the corresponding symbolic composition.

```
A = mx.Variable(:A)
B = mx.Variable(:B)
C = A .* B
a = mx.ones(3) * 4
b = mx.ones(3) * 2
c_exec = mx.bind(C, context=mx.cpu(), args=Dict(:A => a, :B => b))

mx.forward(c_exec)
copy(c_exec.outputs[1])  # copy turns NDArray into Julia Array
# =>
# 3-element Array{Float32,1}:
#  8.0
#  8.0
#  8.0
```

For neural networks, it is easier to use `simple_bind`. By providing the shape for input arguments, it will perform a shape inference for the rest of the arguments and create the `NDArray` automatically. In practice, the binding and executing steps are hidden under the `Model` interface.

TODO Provide pointers to model tutorial and further details about binding and symbolic API.

4.4 High Level Interface

The high level interface include model training and prediction API, etc.

CHAPTER 5

FAQ

5.1 Running MXNet on AWS GPU instances

See the discussions and notes here.

CHAPTER 6

Context

class Context

A context describes the device type and id on which computation should be carried on.

cpu(*dev_id=0*)

Parameters **dev_id** (*Int*) – the CPU id.

Get a CPU context with a specific id. `cpu()` is usually the default context for many operations when no context is specified.

gpu(*dev_id=0*)

Parameters **dev_id** (*Int*) – the GPU device id.

Get a GPU context with a specific id. The K GPUs on a node is typically numbered as 0,. . . ,K-1.

CHAPTER 7

Models

The model API provides convenient high-level interface to do training and predicting on a network described using the symbolic API.

class AbstractModel
The abstract super type of all models in MXNet.jl.

class FeedForward
The feedforward model provides convenient interface to train and predict on feedforward architectures like multi-layer MLP, ConvNets, etc. There is no explicitly handling of *time index*, but it is relatively easy to implement unrolled RNN / LSTM under this framework (**TODO**: add example). For models that handles sequential data explicitly, please use **TODO**...

FeedForward(*arch :: SymbolicNode*, *ctx*)

Parameters

- **arch** – the architecture of the network constructed using the symbolic API.
- **ctx** – the devices on which this model should do computation. It could be a single `Context` or a list of `Context` objects. In the latter case, data parallelization will be used for training. If no context is provided, the default context `cpu()` will be used.

init_model(*self*, *initializer; overwrite=false*, *input_shapes...*)
Initialize the weights in the model.

This method will be called automatically when training a model. So there is usually no need to call this method unless one needs to inspect a model with only randomly initialized weights.

Parameters

- **self** (*FeedForward*) – the model to be initialized.
- **initializer** (*AbstractInitializer*) – an initializer describing how the weights should be initialized.
- **overwrite** (*Bool*) – keyword argument, force initialization even when weights already exists.

- **input_shapes** – the shape of all data and label inputs to this model, given as keyword arguments. For example, `data=(28,28,1,100), label=(100,)`.

predict (*self*, *data; overwrite=false*, *callback=nothing*)

Predict using an existing model. The model should be already initialized, or trained or loaded from a checkpoint. There is an overloaded function that allows to pass the callback as the first argument, so it is possible to do

```
predict(model, data) do batch_output
  # consume or write batch_output to file
end
```

Parameters

- **self** (*FeedForward*) – the model.
- **data** (*AbstractDataProvider*) – the data to perform prediction on.
- **overwrite** (*Bool*) – an `Executor` is initialized the first time predict is called. The memory allocation of the `Executor` depends on the mini-batch size of the test data provider. If you call predict twice with data provider of the same batch-size, then the executor can be potentially be re-used. So, if `overwrite` is false, we will try to re-use, and raise an error if batch-size changed. If `overwrite` is true (the default), a new `Executor` will be created to replace the old one.

Note: Prediction is computationally much less costly than training, so the bottleneck sometimes becomes the IO for copying mini-batches of data. Since there is no concern about convergence in prediction, it is better to set the mini-batch size as large as possible (limited by your device memory) if prediction speed is a concern.

For the same reason, currently prediction will only use the first device even if multiple devices are provided to construct the model.

Note: If you perform further after prediction. The weights are not automatically synchronized if `overwrite` is set to false and the old predictor is re-used. In this case setting `overwrite` to true (the default) will re-initialize the predictor the next time you call predict and synchronize the weights again.

Seealso `train()`, `fit()`, `init_model()`, `load_checkpoint()`

train (*model :: FeedForward*, *...*)

Alias to `fit()`.

fit (*model :: FeedForward*, *optimizer*, *data; kwargs...*)

Train the `model` on `data` with the `optimizer`.

Parameters

- **model** (*FeedForward*) – the model to be trained.
- **optimizer** (*AbstractOptimizer*) – the optimization algorithm to use.
- **data** (*AbstractDataProvider*) – the training data provider.
- **n_epoch** (*Int*) – default 10, the number of full data-passes to run.
- **eval_data** (*AbstractDataProvider*) – keyword argument, default `nothing`. The data provider for the validation set.

- **eval_metric** (*AbstractEvalMetric*) – keyword argument, default `Accuracy()`. The metric used to evaluate the training performance. If `eval_data` is provided, the same metric is also calculated on the validation set.
- **kvstore** (`KVStore` or `Base.Symbol`) – keyword argument, default `:local`. The key-value store used to synchronize gradients and parameters when multiple devices are used for training.
- **initializer** (*AbstractInitializer*) – keyword argument, default `UniformInitializer(0.01)`.
- **force_init** (*Bool*) – keyword argument, default false. By default, the random initialization using the provided `initializer` will be skipped if the model weights already exists, maybe from a previous call to `train()` or an explicit call to `init_model()` or `load_checkpoint()`. When this option is set, it will always do random initialization at the begining of training.
- **callbacks** (`Vector{AbstractCallback}`) – keyword argument, default `[]`. Callbacks to be invoked at each epoch or mini-batch, see `AbstractCallback`.

CHAPTER 8

Initializers

8.1 Interface

class AbstractInitializer
The abstract base class for all initializers.

To define a new initializer, it is enough to derive a new type, and implement one or more of the following methods:

_init_weight (*self :: AbstractInitializer*, *name :: Base.Symbol*, *array :: NDArray*)

_init_bias (*self :: AbstractInitializer*, *name :: Base.Symbol*, *array :: NDArray*)

_init_gamma (*self :: AbstractInitializer*, *name :: Base.Symbol*, *array :: NDArray*)

_init_beta (*self :: AbstractInitializer*, *name :: Base.Symbol*, *array :: NDArray*)

Or, if full behavior customization is needed, override the following function

init (*self :: AbstractInitializer*, *name :: Base.Symbol*, *array :: NDArray*)

8.2 Built-in initializers

class UniformInitializer
Initialize weights according to a uniform distribution within the provided scale.

class NormalInitializer
Initialize weights according to a univariate Gaussian distribution.

NormalIninitializer (*; mu=0*, *sigma=0.01*)
Construct a `NormalInitializer` with mean `mu` and variance `sigma`.

class XavierInitializer
The initializer documented in the paper [Bengio and Glorot 2010]: *Understanding the difficulty of training deep feedforward neuralnetworks*.

There are several different version of the XavierInitializer used in the wild. The general idea is that the variance of the initialization distribution is controlled by the dimensionality of the input and output. As a distribution one can either choose a normal distribution with $\mu = 0$ and σ^2 or a uniform distribution from -σ to σ.

Several different ways of calculating the variance are given in the literature or are used by various libraries.

- [Bengio and Glorot 2010]: `mx.XavierInitializer(distribution = mx.xv_uniform, regularization = mx.xv_avg, magnitude = 1)`
- [K. He, X. Zhang, S. Ren, and J. Sun 2015]: `mx.XavierInitializer(distribution = mx.xv_gaussian, regularization = mx.xv_in, magnitude = 2)`
- caffe_avg: `mx.XavierInitializer(distribution = mx.xv_uniform, regularization = mx.xv_avg, magnitude = 3)`

CHAPTER 9

Optimizers

9.1 Common interfaces

class AbstractOptimizer

Base type for all optimizers.

class AbstractLearningRateScheduler

Base type for all learning rate scheduler.

class AbstractMomentumScheduler

Base type for all momentum scheduler.

class OptimizationState

batch_size

The size of the mini-batch used in stochastic training.

curr_epoch

The current epoch count. Epoch 0 means no training yet, during the first pass through the data, the epoch will be 1; during the second pass, the epoch count will be 1, and so on.

curr_batch

The current mini-batch count. The batch count is reset during every epoch. The batch count 0 means the beginning of each epoch, with no mini-batch seen yet. During the first mini-batch, the mini-batch count will be 1.

curr_iter

The current iteration count. One iteration corresponds to one mini-batch, but unlike the mini-batch count, the iteration count does **not** reset in each epoch. So it track the *total* number of mini-batches seen so far.

get_learning_rate (*scheduler*, *state*)

Parameters

- **scheduler** (*AbstractLearningRateScheduler*) – a learning rate scheduler.

- **state** (*OptimizationState*) – the current state about epoch, mini-batch and iteration count.

Returns the current learning rate.

class LearningRate.**Fixed**

Fixed learning rate scheduler always return the same learning rate.

class LearningRate.**Exp**

$\eta_t = \eta_0\gamma^t$. Here t is the epoch count, or the iteration count if `decay_on_iteration` is set to true.

class LearningRate.**Inv**

$\eta_t = \eta_0 * (1 + \gamma * t)^{(} - power)$. Here t is the epoch count, or the iteration count if `decay_on_iteration` is set to true.

get_momentum(*scheduler*, *state*)

Parameters

- **scheduler** (*AbstractMomentumScheduler*) – the momentum scheduler.
- **state** (*OptimizationState*) – the state about current epoch, mini-batch and iteration count.

Returns the current momentum.

class Momentum.**Null**

The null momentum scheduler always returns 0 for momentum. It is also used to explicitly indicate momentum should not be used.

class Momentum.**Fixed**

Fixed momentum scheduler always returns the same value.

get_updater(*optimizer*)

Parameters **optimizer** (*AbstractOptimizer*) – the underlying optimizer.

A utility function to create an updater function, that uses its closure to store all the states needed for each weights.

9.2 Built-in optimizers

class AbstractOptimizerOptions

Base class for all optimizer options.

normalized_gradient(*opts*, *state*, *grad*)

Parameters

- **opts** (*AbstractOptimizerOptions*) – options for the optimizer, should contain the field `grad_scale`, `grad_clip` and `weight_decay`.
- **state** (*OptimizationState*) – the current optimization state.
- **weight** (*NDArray*) – the trainable weights.
- **grad** (*NDArray*) – the original gradient of the weights.

Get the properly normalized gradient (re-scaled and clipped if necessary).

class SGD

Stochastic gradient descent optimizer.

SGD(*; kwargs...*)

Parameters

- **lr** (*Real*) – default *0.01*, learning rate.
- **lr_scheduler** (*AbstractLearningRateScheduler*) – default *nothing*, a dynamic learning rate scheduler. If set, will overwrite the *lr* parameter.
- **momentum** (*Real*) – default *0.0*, the momentum.
- **momentum_scheduler** (*AbstractMomentumScheduler*) – default *nothing*, a dynamic momentum scheduler. If set, will overwrite the *momentum* parameter.
- **grad_clip** (*Real*) – default *0*, if positive, will clip the gradient into the bounded range *[-grad_clip, grad_clip]*.
- **weight_decay** (*Real*) – default *0.0001*, weight decay is equivalent to adding a global l2 regularizer to the parameters.

class ADAM

The solver described in Diederik Kingma, Jimmy Ba: *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs.LG].

ADAM(*; kwargs...*)

Parameters

- **lr** (*Real*) – default *0.001*, learning rate.
- **lr_scheduler** (*AbstractLearningRateScheduler*) – default *nothing*, a dynamic learning rate scheduler. If set, will overwrite the *lr* parameter.
- **beta1** (*Real*) – default *0.9*.
- **beta2** (*Real*) – default *0.999*.
- **epsilon** (*Real*) – default *1e-8*.
- **grad_clip** (*Real*) – default *0*, if positive, will clip the gradient into the range *[-grad_clip, grad_clip]*.
- **weight_decay** (*Real*) – default *0.00001*, weight decay is equivalent to adding a global l2 regularizer for all the parameters.

CHAPTER 10

Callbacks in training

class AbstractCallback

Abstract type of callback functions used in training.

class AbstractBatchCallback

Abstract type of callbacks to be called every mini-batch.

class AbstractEpochCallback

Abstract type of callbacks to be called every epoch.

every_n_batch(*callback :: Function, n :: Int; call_on_0 = false*)

A convenient function to construct a callback that runs every `n` mini-batches.

Parameters **call_on_0** (*Int*) – keyword argument, default false. Unless set, the callback will **not** be run on batch 0.

For example, the `speedometer()` callback is defined as

```
every_n_iter(frequency, call_on_0=true) do state :: OptimizationState
  if state.curr_batch == 0
    # reset timer
  else
    # compute and print speed
  end
end
```

Seealso `every_n_epoch()`, `speedometer()`.

speedometer(*; frequency=50*)

Create an `AbstractBatchCallback` that measure the training speed (number of samples processed per second) every k mini-batches.

Parameters **frequency** (*Int*) – keyword argument, default 50. The frequency (number of min-batches) to measure and report the speed.

every_n_epoch(*callback :: Function, n :: Int; call_on_0 = false*)

A convenient function to construct a callback that runs every `n` full data-passes.

Parameters **`call_on_0`** (*`Int`*) – keyword argument, default false. Unless set, the callback will **not** be run on epoch 0. Epoch 0 means no training has been performed yet. This is useful if you want to inspect the randomly initialized model that has not seen any data yet.

Seealso `every_n_iter()`.

`do_checkpoint` (*prefix; frequency=1, save_epoch_0=false*)

Create an `AbstractEpochCallback` that save checkpoints of the model to disk. The checkpoints can be loaded back later on.

Parameters

- **`prefix`** (*`AbstractString`*) – the prefix of the filenames to save the model. The model architecture will be saved to prefix-symbol.json, while the weights will be saved to prefix-0012.params, for example, for the 12-th epoch.
- **`frequency`** (*`Int`*) – keyword argument, default 1. The frequency (measured in epochs) to save checkpoints.
- **`save_epoch_0`** (*`Bool`*) – keyword argument, default false. Whether we should save a checkpoint for epoch 0 (model initialized but not seen any data yet).

CHAPTER 11

Evaluation Metrics

Evaluation metrics provide a way to evaluate the performance of a learned model. This is typically used during training to monitor performance on the validation set.

class AbstractEvalMetric

The base class for all evaluation metrics. The sub-types should implement the following interfaces.

update! (*metric*, *labels*, *preds*)

Update and accumulate metrics.

Parameters

- **metric** (*AbstractEvalMetric*) – the metric object.
- **labels** (*Vector{NDArray}*) – the labels from the data provider.
- **preds** (*Vector{NDArray}*) – the outputs (predictions) of the network.

reset! (*metric*)

Reset the accumulation counter.

get (*metric*)

Get the accumulated metrics.

Returns `Vector{Tuple{Base.Symbol, Real}}`, a list of name-value pairs. For example, `[(:accuracy, 0.9)]`.

class Accuracy

Multiclass classification accuracy.

Calculates the mean accuracy per sample for softmax in one dimension. For a multi-dimensional softmax the mean accuracy over all dimensions is calculated.

class MSE

Mean Squared Error. TODO: add support for multi-dimensional outputs.

Calculates the mean squared error regression loss in one dimension.

CHAPTER 12

Data Providers

12.1 Interface

Data providers are wrappers that load external data, be it images, text, or general tensors, and split it into mini-batches so that the model can consume the data in a uniformed way.

class AbstractDataProvider

The root type for all data provider. A data provider should implement the following interfaces:

get_batch_size(*provider*) → Int

Parameters **provider** (*AbstractDataProvider*) – the data provider.

Returns the mini-batch size of the provided data. All the provided data should have the same mini-batch size (i.e. the last dimension).

provide_data(*provider*) → Vector{Tuple{Base.Symbol, Tuple}}

Parameters **provider** (*AbstractDataProvider*) – the data provider.

Returns a vector of (name, shape) pairs describing the names of the data it provides, and the corresponding shapes.

provide_label(*provider*) → Vector{Tuple{Base.Symbol, Tuple}}

Parameters **provider** (*AbstractDataProvider*) – the data provider.

Returns a vector of (name, shape) pairs describing the names of the labels it provides, and the corresponding shapes.

The difference between *data* and *label* is that during training stage, both *data* and *label* will be feeded into the model, while during prediction stage, only *data* is loaded. Otherwise, they could be anything, with any names, and of any shapes. The provided data and label names here should match the input names in a target `SymbolicNode`.

A data provider should also implement the Julia iteration interface, in order to allow iterating through the data set. The provider will be called in the following way:

```
for batch in eachbatch(provider)
  data = get_data(provider, batch)
end
```

which will be translated by Julia compiler into

```
state = Base.start(eachbatch(provider))
while !Base.done(provider, state)
  (batch, state) = Base.next(provider, state)
  data = get_data(provider, batch)
end
```

By default, `eachbatch()` simply returns the provider itself, so the iterator interface is implemented on the provider type itself. But the extra layer of abstraction allows us to implement a data provider easily via a Julia `Task` coroutine. See the data provider defined in *the char-lstm example* for an example of using coroutine to define data providers.

The detailed interface functions for the iterator API is listed below:

`Base.`**`eltype`**(*provider*) → AbstractDataBatch

Parameters **`provider`** (*`AbstractDataProvider`*) – the data provider.

Returns the specific subtype representing a data batch. See `AbstractDataBatch`.

`Base.`**`start`**(*provider*) → AbstractDataProviderState

Parameters **`provider`** (*`AbstractDataProvider`*) – the data provider.

This function is always called before iterating into the dataset. It should initialize the iterator, reset the index, and do data shuffling if needed.

`Base.`**`done`**(*provider*, *state*) → Bool

Parameters

- **`provider`** (*`AbstractDataProvider`*) – the data provider.
- **`state`** (*`AbstractDataProviderState`*) – the state returned by `Base.start()` `Base.next()`.

Returns true if there is no more data to iterate in this dataset.

`Base.`**`next`**(*provider) -> (AbstractDataBatch*, *AbstractDataProviderState*)

Parameters **`provider`** (*`AbstractDataProvider`*) – the data provider.

Returns the current data batch, and the state for the next iteration.

Note sometimes you are wrapping an existing data iterator (e.g. the built-in libmxnet data iterator) that is built with a different convention. It might be difficult to adapt to the interfaces stated here. In this case, you can safely assume that

- `Base.start()` will always be called, and called only once before the iteration starts.
- `Base.done()` will always be called at the beginning of every iteration and always be called once.
- If `Base.done()` return true, the iteration will stop, until the next round, again, starting with a call to `Base.start()`.
- `Base.next()` will always be called only once in each iteration. It will always be called after one and only one call to `Base.done()`; but if `Base.done()` returns true, `Base.next()` will not be called.

With those assumptions, it will be relatively easy to adapt any existing iterator. See the implementation of the built-in `MXDataProvider` for example.

Caution: Please do not use the one data provider simultaneously in two different places, either in parallel, or in a nested loop. For example, the behavior for the following code is undefined

```
for batch in data
  # updating the parameters

  # now let's test the performance on the training set
  for b2 in data
    # ...
  end
end
```

class AbstractDataProviderState

Base type for data provider states.

class AbstractDataBatch

Base type for a data mini-batch. It should implement the following interfaces:

count_samples (*provider*, *batch*) → Int

Parameters **batch** (*AbstractDataBatch*) – the data batch object.

Returns the number of samples in this batch. This number should be greater than 0, but less than or equal to the batch size. This is used to indicate at the end of the data set, there might not be enough samples for a whole mini-batch.

get_data (*provider*, *batch*) → Vector{NDArray}

Parameters

- **provider** (*AbstractDataProvider*) – the data provider.
- **batch** (*AbstractDataBatch*) – the data batch object.

Returns

a vector of data in this batch, should be in the same order as declared in `provide_data()`.

The last dimension of each `NDArray` should always match the batch_size, even when `count_samples()` returns a value less than the batch size. In this case, the data provider is free to pad the remaining contents with any value.

get_label (*provider*, *batch*) → Vector{NDArray}

Parameters

- **provider** (*AbstractDataProvider*) – the data provider.
- **batch** (*AbstractDataBatch*) – the data batch object.

Returns a vector of labels in this batch. Similar to `get_data()`.

The following utility functions will be automatically defined.

get (*provider*, *batch*, *name*) → NDArray

Parameters

- **provider** (*AbstractDataProvider*) – the data provider.
- **batch** (*AbstractDataBatch*) – the data batch object.
- **name** (*Base.Symbol*) – the name of the data to get, should be one of the names provided in either `provide_data()` or `provide_label()`.

Returns the corresponding data array corresponding to that name.

`load_data!`(*provider*, *batch*, *targets*)

Parameters

- **`provider`** (*`AbstractDataProvider`*) – the data provider.
- **`batch`** (*`AbstractDataBatch`*) – the data batch object.
- **`targets`** (*`Vector{Vector{SlicedNDArray}}`*) – the targets to load data into.

The targets is a list of the same length as number of data provided by this provider. Each element in the list is a list of `SlicedNDArray`. This list described a spliting scheme of this data batch into different slices, each slice is specified by a slice-ndarray pair, where *slice* specify the range of samples in the mini-batch that should be loaded into the corresponding *ndarray*.

This utility function is used in data parallelization, where a mini-batch is splited and computed on several different devices.

`load_label!`(*provider*, *batch*, *targets*)

Parameters

- **`provider`** (*`AbstractDataProvider`*) – the data provider.
- **`batch`** (*`AbstractDataBatch`*) – the data batch object.
- **`targets`** (*`Vector{Vector{SlicedNDArray}}`*) – the targets to load label into.

The same as `load_data!()`, except that this is for loading labels.

`class DataBatch`

A basic subclass of `AbstractDataBatch`, that implement the interface by accessing member fields.

`class SlicedNDArray`

A alias type of `Tuple{UnitRange{Int},NDArray}`.

12.2 Built-in data providers

`class ArrayDataProvider`

A convenient tool to iterate `NDArray` or Julia `Array`.

`ArrayDataProvider`(*data[, label]; batch_size, shuffle, data_padding, label_padding*)

Construct a data provider from `NDArray` or Julia Arrays.

Parameters

- **`data`** – the data, could be
  - a `NDArray`, or a Julia Array. This is equivalent to `:data => data`.
  - a name-data pair, like `:mydata => array`, where `:mydata` is the name of the data and `array` is an `NDArray` or a Julia Array.
  - a list of name-data pairs.
- **`label`** – the same as the `data` parameter. When this argument is omitted, the constructed provider will provide no labels.
- **`batch_size`** (*`Int`*) – the batch size, default is 0, which means treating the whole array as a single mini-batch.
- **`shuffle`** (*`Bool`*) – turn on if the data should be shuffled at every epoch.

- **data_padding** (*Real*) – when the mini-batch goes beyond the dataset boundary, there might be less samples to include than a mini-batch. This value specify a scalar to pad the contents of all the missing data points.
- **label_padding** (*Real*) – the same as `data_padding`, except for the labels.

TODO: remove `data_padding` and `label_padding`, and implement rollover that copies the last or first several training samples to feed the padding.

12.3 libmxnet data providers

class MXDataProvider

A data provider that wrap built-in data iterators from libmxnet. See below for a list of built-in data iterators.

CSVIter(...)

Can also be called with the alias `CSVProvider`. Create iterator for dataset in csv.

Parameters

- **data_name** (*Base.Symbol*) – keyword argument, default `:data`. The name of the data.
- **label_name** (*Base.Symbol*) – keyword argument, default `:softmax_label`. The name of the label. Could be `nothing` if no label is presented in this dataset.
- **data_csv** (*string, required*) – Dataset Param: Data csv path.
- **data_shape** (*Shape(tuple), required*) – Dataset Param: Shape of the data.
- **label_csv** (*string, optional, default='NULL'*) – Dataset Param: Label csv path. If is NULL, all labels will be returned as 0
- **label_shape** (*Shape(tuple), optional, default=(1,)*) – Dataset Param: Shape of the label.

Returns the constructed `MXDataProvider`.

ImageRecordIter(...)

Can also be called with the alias `ImageRecordProvider`. Create iterator for dataset packed in recordio.

Parameters

- **data_name** (*Base.Symbol*) – keyword argument, default `:data`. The name of the data.
- **label_name** (*Base.Symbol*) – keyword argument, default `:softmax_label`. The name of the label. Could be `nothing` if no label is presented in this dataset.
- **path_imglist** (*string, optional, default=''*) – Dataset Param: Path to image list.
- **path_imgrec** (*string, optional, default='./data/imgrec.rec'*) – Dataset Param: Path to image record file.
- **label_width** (*int, optional, default='1'*) – Dataset Param: How many labels for an image.
- **data_shape** (*Shape(tuple), required*) – Dataset Param: Shape of each instance generated by the DataIter.
- **preprocess_threads** (*int, optional, default='4'*) – Backend Param: Number of thread to do preprocessing.

- **verbose** (*boolean, optional, default=True*) – Auxiliary Param: Whether to output parser information.
- **num_parts** (*int, optional, default='1'*) – partition the data into multiple parts
- **part_index** (*int, optional, default='0'*) – the index of the part will read
- **shuffle** (*boolean, optional, default=False*) – Augmentation Param: Whether to shuffle data.
- **seed** (*int, optional, default='0'*) – Augmentation Param: Random Seed.
- **batch_size** (*int (non-negative), required*) – Batch Param: Batch size.
- **round_batch** (*boolean, optional, default=True*) – Batch Param: Use round robin to handle overflow batch.
- **prefetch_buffer** (*, optional, default=4*) – Backend Param: Number of prefetched parameters
- **rand_crop** (*boolean, optional, default=False*) – Augmentation Param: Whether to random crop on the image
- **crop_y_start** (*int, optional, default='-1'*) – Augmentation Param: Where to nonrandom crop on y.
- **crop_x_start** (*int, optional, default='-1'*) – Augmentation Param: Where to nonrandom crop on x.
- **max_rotate_angle** (*int, optional, default='0'*) – Augmentation Param: rotated randomly in [-max_rotate_angle, max_rotate_angle].
- **max_aspect_ratio** (*float, optional, default=0*) – Augmentation Param: denotes the max ratio of random aspect ratio augmentation.
- **max_shear_ratio** (*float, optional, default=0*) – Augmentation Param: denotes the max random shearing ratio.
- **max_crop_size** (*int, optional, default='-1'*) – Augmentation Param: Maximum crop size.
- **min_crop_size** (*int, optional, default='-1'*) – Augmentation Param: Minimum crop size.
- **max_random_scale** (*float, optional, default=1*) – Augmentation Param: Maxmum scale ratio.
- **min_random_scale** (*float, optional, default=1*) – Augmentation Param: Minimum scale ratio.
- **max_img_size** (*float, optional, default=1e+10*) – Augmentation Param: Maxmum image size after resizing.
- **min_img_size** (*float, optional, default=0*) – Augmentation Param: Minimum image size after resizing.
- **random_h** (*int, optional, default='0'*) – Augmentation Param: Maximum value of H channel in HSL color space.
- **random_s** (*int, optional, default='0'*) – Augmentation Param: Maximum value of S channel in HSL color space.
- **random_l** (*int, optional, default='0'*) – Augmentation Param: Maximum value of L channel in HSL color space.

- **rotate** (*int, optional, default='-1'*) – Augmentation Param: Rotate angle.
- **fill_value** (*int, optional, default='255'*) – Augmentation Param: Maximum value of illumination variation.
- **inter_method** (*int, optional, default='1'*) – Augmentation Param: 0-NN 1-bilinear 2-cubic 3-area 4-lanczos4 9-auto 10-rand.
- **mirror** (*boolean, optional, default=False*) – Augmentation Param: Whether to mirror the image.
- **rand_mirror** (*boolean, optional, default=False*) – Augmentation Param: Whether to mirror the image randomly.
- **mean_img** (*string, optional, default=''*) – Augmentation Param: Mean Image to be subtracted.
- **mean_r** (*float, optional, default=0*) – Augmentation Param: Mean value on R channel.
- **mean_g** (*float, optional, default=0*) – Augmentation Param: Mean value on G channel.
- **mean_b** (*float, optional, default=0*) – Augmentation Param: Mean value on B channel.
- **mean_a** (*float, optional, default=0*) – Augmentation Param: Mean value on Alpha channel.
- **scale** (*float, optional, default=1*) – Augmentation Param: Scale in color space.
- **max_random_contrast** (*float, optional, default=0*) – Augmentation Param: Maximum ratio of contrast variation.
- **max_random_illumination** (*float, optional, default=0*) – Augmentation Param: Maximum value of illumination variation.

Returns the constructed `MXDataProvider`.

MNISTIter(...)

Can also be called with the alias `MNISTProvider`. Create iterator for MNIST hand-written digit number recognition dataset.

Parameters

- **data_name** (*Base.Symbol*) – keyword argument, default `:data`. The name of the data.
- **label_name** (*Base.Symbol*) – keyword argument, default `:softmax_label`. The name of the label. Could be `nothing` if no label is presented in this dataset.
- **image** (*string, optional, default='./train-images-idx3-ubyte'*) – Dataset Param: Mnist image path.
- **label** (*string, optional, default='./train-labels-idx1-ubyte'*) – Dataset Param: Mnist label path.
- **batch_size** (*int, optional, default='128'*) – Batch Param: Batch Size.
- **shuffle** (*boolean, optional, default=True*) – Augmentation Param: Whether to shuffle data.
- **flat** (*boolean, optional, default=False*) – Augmentation Param: Whether to flat the data into 1D.

- **seed** (*int, optional, default='0'*) – Augmentation Param: Random Seed.
- **silent** (*boolean, optional, default=False*) – Auxiliary Param: Whether to print out data info.
- **num_parts** (*int, optional, default='1'*) – partition the data into multiple parts
- **part_index** (*int, optional, default='0'*) – the index of the part will read
- **prefetch_buffer** (*, optional, default=4*) – Backend Param: Number of prefetched parameters

Returns the constructed `MXDataProvider`.

CHAPTER 13

NDArray API

class NDArray

Wrapper of the `NDArray` type in `libmxnet`. This is the basic building block of tensor-based computation.

> **Note:** since C/C++ use row-major ordering for arrays while Julia follows a column-major ordering. To keep things consistent, we keep the underlying data in their original layout, but use *language-native* convention when we talk about shapes. For example, a mini-batch of 100 MNIST images is a tensor of C/C++/Python shape (100,1,28,28), while in Julia, the same piece of memory have shape (28,28,1,100).

context (*arr :: NDArray*)

Get the context that this `NDArray` lives on.

empty (*shape :: Tuple*, *ctx :: Context*)
empty (*shape :: Tuple*)
empty (*dim1*, *dim2*, *...*)

Allocate memory for an uninitialized `NDArray` with specific shape.

13.1 Interface functions similar to Julia Arrays

zeros (*shape :: Tuple*, *ctx :: Context*)
zeros (*shape :: Tuple*)
zeros (*dim1*, *dim2*, *...*)

Create zero-ed `NDArray` with specific shape.

ones (*shape :: Tuple*, *ctx :: Context*)
ones (*shape :: Tuple*)
ones (*dim1*, *dim2*, *...*)

Create an `NDArray` with specific shape and initialize with 1.

size (*arr :: NDArray*)

size (*arr :: NDArray*, *dim :: Int*)

Get the shape of an `NDArray`. The shape is in Julia's column-major convention. See also the *notes on NDArray shapes*.

length (*arr :: NDArray*)

Get the number of elements in an `NDArray`.

ndims (*arr :: NDArray*)

Get the number of dimensions of an `NDArray`. Is equivalent to `length(size(arr))`.

eltype (*arr :: NDArray*)

Get the element type of an `NDArray`. Currently the element type is always `mx.MX_float`.

slice (*arr :: NDArray*, *start:stop*)

Create a view into a sub-slice of an `NDArray`. Note only slicing at the slowest changing dimension is supported. In Julia's column-major perspective, this is the last dimension. For example, given an `NDArray` of shape (2,3,4), `slice(array, 2:3)` will create a `NDArray` of shape (2,3,2), sharing the data with the original array. This operation is used in data parallelization to split mini-batch into sub-batches for different devices.

setindex! (*arr :: NDArray*, *val*, *idx*)

Assign values to an `NDArray`. Elementwise assignment is not implemented, only the following scenarios are supported

- `arr[:] = val`: whole array assignment, `val` could be a scalar or an array (Julia `Array` or `NDArray`) of the same shape.
- `arr[start:stop] = val`: assignment to a *slice*, `val` could be a scalar or an array of the same shape to the slice. See also `slice()`.

getindex (*arr :: NDArray*, *idx*)

Shortcut for `slice()`. A typical use is to write

```
arr[:] += 5
```

which translates into

```
arr[:] = arr[:] + 5
```

which furthur translates into

```
setindex!(getindex(arr, Colon()), 5, Colon())
```

Note: The behavior is quite different from indexing into Julia's `Array`. For example, `arr[2:5]` create a **copy** of the sub-array for Julia `Array`, while for `NDArray`, this is a *slice* that shares the memory.

13.2 Copying functions

copy! (*dst :: Union{NDArray, Array}*, *src :: Union{NDArray, Array}*)

Copy contents of `src` into `dst`.

copy (*arr :: NDArray*)
copy (*arr :: NDArray*, *ctx :: Context*)
copy (*arr :: Array*, *ctx :: Context*)

Create a copy of an array. When no `Context` is given, create a Julia `Array`. Otherwise, create an `NDArray` on the specified context.

convert (*::Type{Array{T}}, arr :: NDArray*)
Convert an `NDArray` into a Julia `Array` of specific type. Data will be copied.

13.3 Basic arithmetics

@inplace ()
Julia does not support re-definiton of += operator (like `__iadd__` in python), When one write `a += b`, it gets translated to `a = a+b`. `a+b` will allocate new memory for the results, and the newly allocated `NDArray` object is then assigned back to a, while the original contents in a is discarded. This is very inefficient when we want to do inplace update.

This macro is a simple utility to implement this behavior. Write

```
@mx.inplace a += b
```

will translate into

```
mx.add_to!(a, b)
```

which will do inplace adding of the contents of `b` into `a`.

add_to! (*dst :: NDArray, args :: Union{Real, NDArray}...*)
Add a bunch of arguments into `dst`. Inplace updating.

+ (*args...*)
.+ (*args...*)
Summation. Multiple arguments of either scalar or `NDArray` could be added together. Note at least the first or second argument needs to be an `NDArray` to avoid ambiguity of built-in summation.

sub_from! (*dst :: NDArray, args :: Union{Real, NDArray}...*)
Subtract a bunch of arguments from `dst`. Inplace updating.

- (*arg0, arg1*)
- (*arg0*)
.- (*arg0, arg1*)
Subtraction `arg0 - arg1`, of scalar types or `NDArray`. Or create the negative of `arg0`.

mul_to! (*dst :: NDArray, arg :: Union{Real, NDArray}*)
Elementwise multiplication into `dst` of either a scalar or an `NDArray` of the same shape. Inplace updating.

.* (*arg0, arg1*)
Elementwise multiplication of `arg0` and `arg`, could be either scalar or `NDArray`.

***** (*arg0, arg1*)
Currently only multiplication a scalar with an `NDArray` is implemented. Matrix multiplication is to be added soon.

div_from! (*dst :: NDArray, arg :: Union{Real, NDArray}*)
Elementwise divide a scalar or an `NDArray` of the same shape from `dst`. Inplace updating.

./ (*arg0 :: NDArray, arg :: Union{Real, NDArray}*)
Elementwise dividing an `NDArray` by a scalar or another `NDArray` of the same shape.

/ (*arg0 :: NDArray, arg :: Real*)
Divide an `NDArray` by a scalar. Matrix division (solving linear systems) is not implemented yet.

13.4 Manipulating as Julia Arrays

@nd_as_jl(*captures..., statement*)

A convenient macro that allows to operate `NDArray` as Julia Arrays. For example,

```
x = mx.zeros(3,4)
y = mx.ones(3,4)
z = mx.zeros((3,4), mx.gpu())

@mx.nd_as_jl ro=(x,y) rw=z begin
  # now x, y, z are just ordinary Julia Arrays
  z[:,1] = y[:,2]
  z[:,2] = 5
end
```

Under the hood, the macro convert all the declared captures from `NDArray` into Julia Arrays, by using `try_get_shared()`. And automatically commit the modifications back into the `NDArray` that is declared as `rw`. This is useful for fast prototyping and when implement non-critical computations, such as `AbstractEvalMetric`.

Note:

- Multiple `rw` and / or `ro` capture declaration could be made.
- The macro does **not** check to make sure that `ro` captures are not modified. If the original `NDArray` lives in CPU memory, then it is very likely the corresponding Julia Array shares data with the `NDArray`, so modifying the Julia Array will also modify the underlying `NDArray`.
- More importantly, since the `NDArray` is asynchronized, we will wait for *writing* for `rw` variables but wait only for *reading* in `ro` variables. If we write into those `ro` variables, **and** if the memory is shared, racing condition might happen, and the behavior is undefined.
- When an `NDArray` is declared to be captured as `rw`, its contents is always sync back in the end.
- The execution results of the expanded macro is always `nothing`.
- The statements are wrapped in a `let`, thus locally introduced new variables will not be available after the statements. So you will need to declare the variables before calling the macro if needed.

try_get_shared(*arr*)

Try to create a Julia array by sharing the data with the underlying `NDArray`.

Parameters **arr** (*NDArray*) – the array to be shared.

Warning: The returned array does not guarantee to share data with the underlying `NDArray`. In particular, data sharing is possible only when the `NDArray` lives on CPU.

is_shared(*j_arr, arr*)

Test whether `j_arr` is sharing data with `arr`.

Parameters

- **j_arr** (*Array*) – the Julia Array.
- **arr** (*NDArray*) – the `NDArray`.

13.5 IO

load (*filename*, *::Type{NDArray}*)

Load NDArrays from binary file.

Parameters **filename** (*AbstractString*) – the path of the file to load. It could be S3 or HDFS address.

Returns Either `Dict{Base.Symbol, NDArray}` or `Vector{NDArray}`.

If the `libmxnet` is built with the corresponding component enabled. Examples

- `s3://my-bucket/path/my-s3-ndarray`
- `hdfs://my-bucket/path/my-hdfs-ndarray`
- `/path-to/my-local-ndarray`

save (*filename :: AbstractString*, *data*)

Save NDarrays to binary file. Filename could be S3 or HDFS address, if `libmxnet` is built with corresponding support.

Parameters

- **filename** (*AbstractString*) – path to the binary file to write to.
- **data** (`NDArray`, or a `Vector{NDArray}` or a `Dict{Base.Symbol, NDArray}`.) – data to save to file.

13.6 libmxnet APIs

The libxmnet APIs are automatically imported from `libmxnet.so`. The functions listed here operate on `NDArray` objects. The arguments to the functions are typically ordered as

```
func_name(arg_in1, arg_in2, ..., scalar1, scalar2, ..., arg_out1, arg_out2, ...)
```

unless `NDARRAY_ARG_BEFORE_SCALAR` is not set. In this case, the scalars are put before the input arguments:

```
func_name(scalar1, scalar2, ..., arg_in1, arg_in2, ..., arg_out1, arg_out2, ...)
```

If `ACCEPT_EMPTY_MUTATE_TARGET` is set. An overloaded function without the output arguments will also be defined:

```
func_name(arg_in1, arg_in2, ..., scalar1, scalar2, ...)
```

Upon calling, the output arguments will be automatically initialized with empty NDArrays.

Those functions always return the output arguments. If there is only one output (the typical situation), that object (`NDArray`) is returned. Otherwise, a tuple containing all the outputs will be returned.

13.6.1 Public APIs

abs (...)

Take absolute value of the src

Parameters **src** (*NDArray*) – Source input to the function

argmax_channel(...)

Take argmax indices of each channel of the src.The result will be ndarray of shape (num_channel,) on the same device.

Parameters **src** (*NDArray*) – Source input to the function

ceil(...)

Take ceil value of the src

Parameters **src** (*NDArray*) – Source input to the function

choose_element_0index(...)

Choose one element from each line(row for python, column for R/Julia) in lhs according to index indicated by rhs. This function assume rhs uses 0-based index.

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*NDArray*) – Right operand to the function.

clip(...)

Clip ndarray elements to range (a_min, a_max)

Parameters

- **src** (*NDArray*) – Source input
- **a_min** (*real_t*) – Minimum value
- **a_max** (*real_t*) – Maximum value

cos(...)

Take cos of the src

Parameters **src** (*NDArray*) – Source input to the function

dot(...)

Calculate 2D matrix multiplication

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*NDArray*) – Right operand to the function.

exp(...)

Take exp of the src

Parameters **src** (*NDArray*) – Source input to the function

fill_element_0index(...)

Fill one element of each line(row for python, column for R/Julia) in lhs according to index indicated by rhs and values indicated by mhs. This function assume rhs uses 0-based index.

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **mhs** (*NDArray*) – Middle operand to the function.
- **rhs** (*NDArray*) – Right operand to the function.

floor(...)

Take floor value of the src

Parameters **src** (*NDArray*) – Source input to the function

log(...)
Take log of the src

Parameters **src** (*NDArray*) – Source input to the function

max(...)
Take max of the src.The result will be ndarray of shape (1,) on the same device.

Parameters **src** (*NDArray*) – Source input to the function

min(...)
Take min of the src.The result will be ndarray of shape (1,) on the same device.

Parameters **src** (*NDArray*) – Source input to the function

norm(...)
Take L2 norm of the src.The result will be ndarray of shape (1,) on the same device.

Parameters **src** (*NDArray*) – Source input to the function

round(...)
Take round value of the src

Parameters **src** (*NDArray*) – Source input to the function

rsqrt(...)
Take rsqrt of the src

Parameters **src** (*NDArray*) – Source input to the function

sign(...)
Take sign value of the src

Parameters **src** (*NDArray*) – Source input to the function

sin(...)
Take sin of the src

Parameters **src** (*NDArray*) – Source input to the function

sqrt(...)
Take sqrt of the src

Parameters **src** (*NDArray*) – Source input to the function

square(...)
Take square of the src

Parameters **src** (*NDArray*) – Source input to the function

sum(...)
Take sum of the src.The result will be ndarray of shape (1,) on the same device.

Parameters **src** (*NDArray*) – Source input to the function

13.6.2 Internal APIs

Note: Document and signatures for internal API functions might be incomplete.

_copyto(...)

Parameters **src** (*NDArray*) – Source input to the function.

_div(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*NDArray*) – Right operand to the function.

_div_scalar(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*real_t*) – Right operand to the function.

_imdecode(...)

Decode an image, clip to (x0, y0, x1, y1), substract mean, and write to buffer

Parameters

- **mean** (*NDArray*) – image mean
- **index** (*int*) – buffer position for output
- **x0** (*int*) – x0
- **y0** (*int*) – y0
- **x1** (*int*) – x1
- **y1** (*int*) – y1
- **c** (*int*) – channel
- **size** (*int*) – length of str_img

_minus(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*NDArray*) – Right operand to the function.

_minus_scalar(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*real_t*) – Right operand to the function.

_mul(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*NDArray*) – Right operand to the function.

_mul_scalar(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*real_t*) – Right operand to the function.

_onehot_encode(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*NDArray*) – Right operand to the function.

_plus(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*NDArray*) – Right operand to the function.

_plus_scalar(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*real_t*) – Right operand to the function.

_random_gaussian(...)

_random_uniform(...)

_rdiv_scalar(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*real_t*) – Right operand to the function.

_rminus_scalar(...)

Parameters

- **lhs** (*NDArray*) – Left operand to the function.
- **rhs** (*real_t*) – Right operand to the function.

_set_value(...)

Parameters **src** (*real_t*) – Source input to the function.

CHAPTER 14

Symbolic API

class SymbolicNode
SymbolicNode is the basic building block of the symbolic graph in MXNet.jl.

deepcopy(*self :: SymbolicNode*)
Make a deep copy of a SymbolicNode.

copy(*self :: SymbolicNode*)
Make a copy of a SymbolicNode. The same as making a deep copy.

call(*self :: SymbolicNode*, *args :: SymbolicNode...*)
call(*self :: SymbolicNode; kwargs...*)
Make a new node by composing `self` with `args`. Or the arguments can be specified using keyword arguments.

list_arguments(*self :: SymbolicNode*)
List all the arguments of this node. The argument for a node contains both the inputs and parameters. For example, a `FullyConnected` node will have both data and weights in its arguments. A composed node (e.g. a MLP) will list all the arguments for intermediate nodes.

Returns A list of symbols indicating the names of the arguments.

list_outputs(*self :: SymbolicNode*)
List all the outputs of this node.

Returns A list of symbols indicating the names of the outputs.

list_auxiliary_states(*self :: SymbolicNode*)
List all auxiliary states in the symbool.

Auxiliary states are special states of symbols that do not corresponds to an argument, and do not have gradient. But still be useful for the specific operations. A common example of auxiliary state is the moving_mean and moving_variance in BatchNorm. Most operators do not have Auxiliary states.

Returns A list of symbols indicating the names of the auxiliary states.

get_internals(*self :: SymbolicNode*)
Get a new grouped `SymbolicNode` whose output contains all the internal outputs of this `SymbolicNode`.

get_attr(*self :: SymbolicNode, key :: Symbol*)

Get attribute attached to this SymbolicNode belonging to key. :return: The value belonging to key as a Nullable.

set_attr(*self:: SymbolicNode, key :: Symbol, value :: AbstractString*)

Set the attribute key to value for this SymbolicNode.

> **Warning:** It is encouraged not to call this function directly, unless you know exactly what you are doing. The recommended way of setting attributes is when creating the SymbolicNode. Changing the attributes of a SymbolicNode that is already been used somewhere else might cause unexpected behavior and inconsistency.

Variable(*name :: Union{Symbol, AbstractString}*)

Create a symbolic variable with the given name. This is typically used as a placeholder. For example, the data node, acting as the starting point of a network architecture.

Parameters **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this Variable.

Group(*nodes :: SymbolicNode...*)

Create a SymbolicNode by grouping nodes together.

infer_shape(*self :: SymbolicNode; args...*)
infer_shape(*self :: SymbolicNode; kwargs...*)

Do shape inference according to the input shapes. The input shapes could be provided as a list of shapes, which should specify the shapes of inputs in the same order as the arguments returned by `list_arguments()`. Alternatively, the shape information could be specified via keyword arguments.

Returns A 3-tuple containing shapes of all the arguments, shapes of all the outputs and shapes of all the auxiliary variables. If shape inference failed due to incomplete or incompatible inputs, the return value will be `(nothing, nothing, nothing)`.

getindex(*self :: SymbolicNode, idx :: Union{Int, Base.Symbol, AbstractString}*)

Get a node representing the specified output of this node. The index could be a symbol or string indicating the name of the output, or a 1-based integer indicating the index, as in the list of `list_outputs()`.

to_json(*self :: SymbolicNode*)

Convert a SymbolicNode into a JSON string.

from_json(*repr :: AbstractString, ::Type{SymbolicNode}*)

Load a SymbolicNode from a JSON string representation.

load(*filename :: AbstractString, ::Type{SymbolicNode}*)

Load a SymbolicNode from a JSON file.

save(*filename :: AbstractString, node :: SymbolicNode*)

Save a SymbolicNode to a JSON file.

14.1 libmxnet APIs

14.1.1 Public APIs

Activation(*...*)

Apply activation function to input.Softmax Activation is only available with CUDNN on GPUand will be computed at each location across channel if input is 4D.

Parameters

- **data** (*SymbolicNode*) – Input data to activation function.
- **act_type** (*{'relu', 'sigmoid', 'softrelu', 'tanh'}, required*) – Activation function to be applied.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

BatchNorm(...)

Apply batch normalization to input.

Parameters

- **data** (*SymbolicNode*) – Input data to batch normalization
- **eps** (*float, optional, default=0.001*) – Epsilon to prevent div 0
- **momentum** (*float, optional, default=0.9*) – Momentum for moving average
- **fix_gamma** (*boolean, optional, default=True*) – Fix gamma while training
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

BlockGrad(...)

Get output from a symbol and pass 0 gradient back

Parameters

- **data** (*SymbolicNode*) – Input data.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

Cast(...)

Cast array to a different data type.

Parameters

- **data** (*SymbolicNode*) – Input data to cast function.
- **dtype** (*{'float16', 'float32', 'float64', 'int32', 'uint8'}, required*) – Target data type.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

Concat(...)

Perform an feature concat on channel dim (dim 1) over all the inputs.

This function support variable length positional `SymbolicNode` inputs.

Parameters

- **data** (*SymbolicNode[]*) – List of tensors to concatenate
- **num_args** (*int, required*) – Number of inputs to be concated.
- **dim** (*int, optional, default='1'*) – the dimension to be concated.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

Convolution(...)

Apply convolution to input then add a bias.

Parameters

- **data** (*SymbolicNode*) – Input data to the ConvolutionOp.
- **weight** (*SymbolicNode*) – Weight matrix.
- **bias** (*SymbolicNode*) – Bias parameter.
- **kernel** (*Shape(tuple), required*) – convolution kernel size: (y, x)
- **stride** (*Shape(tuple), optional, default=(1,1)*) – convolution stride: (y, x)
- **dilate** (*Shape(tuple), optional, default=(1,1)*) – convolution dilate: (y, x)
- **pad** (*Shape(tuple), optional, default=(0,0)*) – pad for convolution: (y, x)
- **num_filter** (*int (non-negative), required*) – convolution filter(channel) number
- **num_group** (*int (non-negative), optional, default=1*) – Number of groups partition. This option is not supported by CuDNN, you can use SliceChannel to num_group,apply convolution and concat instead to achieve the same need.
- **workspace** (*long (non-negative), optional, default=512*) – Tmp workspace for convolution (MB).
- **no_bias** (*boolean, optional, default=False*) – Whether to disable bias parameter.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

Crop(...)

Crop the 2nd and 3rd dim of input data, with the corresponding size of w_h or with width and height of the second input symbol

This function support variable length positional `SymbolicNode` inputs.

Parameters

- **num_args** (*int, required*) – Number of inputs for crop, if equals one, then we will use the h_wfor crop heihgt and width, else if equals two, then we will use the heightand width of the second input symbol, we name crop_like here
- **offset** (*Shape(tuple), optional, default=(0,0)*) – crop offset coordinate: (y, x)
- **h_w** (*Shape(tuple), optional, default=(0,0)*) – crop height and weight: (h, w)
- **center_crop** (*boolean, optional, default=False*) – If set to true, then it will use be the center_crop,or it will crop using the shape of crop_like
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

Deconvolution(...)

Apply deconvolution to input then add a bias.

Parameters

- **data** (*SymbolicNode*) – Input data to the DeconvolutionOp.
- **weight** (*SymbolicNode*) – Weight matrix.
- **bias** (*SymbolicNode*) – Bias parameter.
- **kernel** (*Shape(tuple), required*) – deconvolution kernel size: (y, x)
- **stride** (*Shape(tuple), optional, default=(1,1)*) – deconvolution stride: (y, x)
- **pad** (*Shape(tuple), optional, default=(0,0)*) – pad for deconvolution: (y, x)
- **num_filter** (*int (non-negative), required*) – deconvolution filter(channel) number
- **num_group** (*int (non-negative), optional, default=1*) – number of groups partition
- **workspace** (*long (non-negative), optional, default=512*) – Tmp workspace for deconvolution (MB)
- **no_bias** (*boolean, optional, default=True*) – Whether to disable bias parameter.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

Dropout(...)

Apply dropout to input

Parameters

- **data** (*SymbolicNode*) – Input data to dropout.

- **p** (*float, optional, default=0.5*) – Fraction of the input that gets dropped out at training time
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

ElementWiseSum(...)

Perform an elementwise sum over all the inputs.

This function support variable length positional `SymbolicNode` inputs.

Parameters

- **num_args** (*int, required*) – Number of inputs to be summed.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

Embedding(...)

Get embedding for one-hot input

Parameters

- **data** (*SymbolicNode*) – Input data to the EmbeddingOp.
- **weight** (*SymbolicNode*) – Enbedding weight matrix.
- **input_dim** (*int, required*) – input dim of one-hot encoding
- **output_dim** (*int, required*) – output dim of embedding
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

Flatten(...)

Flatten input

Parameters

- **data** (*SymbolicNode*) – Input data to flatten.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

FullyConnected(...)

Apply matrix multiplication to input then add a bias.

Parameters

- **data** (*SymbolicNode*) – Input data to the FullyConnectedOp.
- **weight** (*SymbolicNode*) – Weight matrix.

- **bias** (*SymbolicNode*) – Bias parameter.
- **num_hidden** (*int, required*) – Number of hidden nodes of the output.
- **no_bias** (*boolean, optional, default=False*) – Whether to disable bias parameter.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

IdentityAttachKLSparseReg (...)

Apply a sparse regularization to the output a sigmoid activation function.

Parameters

- **data** (*SymbolicNode*) – Input data.
- **sparseness_target** (*float, optional, default=0.1*) – The sparseness target
- **penalty** (*float, optional, default=0.001*) – The tradeoff parameter for the sparseness penalty
- **momentum** (*float, optional, default=0.9*) – The momentum for running average
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

LRN (...)

Apply convolution to input then add a bias.

Parameters

- **data** (*SymbolicNode*) – Input data to the ConvolutionOp.
- **alpha** (*float, optional, default=0.0001*) – value of the alpha variance scaling parameter in the normalization formula
- **beta** (*float, optional, default=0.75*) – value of the beta power parameter in the normalization formula
- **knorm** (*float, optional, default=2*) – value of the k parameter in normalization formula
- **nsize** (*int (non-negative), required*) – normalization window width in elements.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

LeakyReLU (...)

Apply activation function to input.

Parameters

- **data** (*SymbolicNode*) – Input data to activation function.
- **act_type** (*{'elu', 'leaky', 'prelu', 'rrelu'},optional, default='leaky'*) – Activation function to be applied.
- **slope** (*float, optional, default=0.25*) – Init slope for the activation. (For leaky and elu only)
- **lower_bound** (*float, optional, default=0.125*) – Lower bound of random slope. (For rrelu only)
- **upper_bound** (*float, optional, default=0.334*) – Upper bound of random slope. (For rrelu only)
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

LinearRegressionOutput(...)

Use linear regression for final output, this is used on final output of a net.

Parameters

- **data** (*SymbolicNode*) – Input data to function.
- **label** (*SymbolicNode*) – Input label to function.
- **grad_scale** (*float, optional, default=1*) – Scale the gradient by a float factor
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

LogisticRegressionOutput(...)

Use Logistic regression for final output, this is used on final output of a net. Logistic regression is suitable for binary classification or probability prediction tasks.

Parameters

- **data** (*SymbolicNode*) – Input data to function.
- **label** (*SymbolicNode*) – Input label to function.
- **grad_scale** (*float, optional, default=1*) – Scale the gradient by a float factor
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

MAERegressionOutput(...)

Use mean absolute error regression for final output, this is used on final output of a net.

Parameters

- **data** (*SymbolicNode*) – Input data to function.

- **label** (*SymbolicNode*) – Input label to function.
- **grad_scale** (*float, optional, default=1*) – Scale the gradient by a float factor
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

Pooling (...)

Perform spatial pooling on inputs.

Parameters

- **data** (*SymbolicNode*) – Input data to the pooling operator.
- **kernel** (*Shape(tuple), required*) – pooling kernel size: (y, x)
- **pool_type** (*{'avg', 'max', 'sum'}, required*) – Pooling type to be applied.
- **stride** (*Shape(tuple), optional, default=(1,1)*) – stride: for pooling (y, x)
- **pad** (*Shape(tuple), optional, default=(0,0)*) – pad for pooling: (y, x)
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

Reshape (...)

Reshape input to target shape

Parameters

- **data** (*SymbolicNode*) – Input data to reshape.
- **target_shape** (*Shape(tuple), required*) – Target new shape. One and only one dim can be 0, in which case it will be inferred from the rest of dims
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

SliceChannel (...)

Slice channel into many outputs with equally divided channel

Parameters

- **num_outputs** (*int, required*) – Number of outputs to be sliced.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode[].

Softmax (...)

DEPRECATED: Perform a softmax transformation on input. Please use SoftmaxOutput

Parameters

- **`data`** (*`SymbolicNode`*) – Input data to softmax.
- **`grad_scale`** (*`float, optional, default=1`*) – Scale the gradient by a float factor
- **`ignore_label`** (*`float, optional, default=-1`*) – the ignore_label will not work in backward, and this only be used when multi_output=true
- **`multi_output`** (*`boolean, optional, default=False`*) – If set to true, for a (n,k,x_1,..,x_n) dimensional input tensor, softmax will generate n*x_1*. . . *x_n output, each has k classes
- **`use_ignore`** (*`boolean, optional, default=False`*) – If set to true, the ignore_label value will not contribute to the backward gradient
- **`name`** (*`Symbol`*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **`AbstractString} attrs`** (*`Dict{Symbol,`*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

`SoftmaxActivation`(...)

Apply softmax activation to input. This is intended for internal layers. For output (loss layer) please use SoftmaxOutput. If type=instance, this operator will compute a softmax for each instance in the batch; this is the default mode. If type=channel, this operator will compute a num_channel-class softmax at each position of each instance; this can be used for fully convolutional network, image segmentation, etc.

Parameters

- **`data`** (*`SymbolicNode`*) – Input data to activation function.
- **`type`** (*`{'channel', 'instance'},optional, default='instance'`*) – Softmax Mode. If set to instance, this operator will compute a softmax for each instance in the batch; this is the default mode. If set to channel, this operator will compute a num_channel-class softmax at each position of each instance; this can be used for fully convolutional network, image segmentation, etc.
- **`name`** (*`Symbol`*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **`AbstractString} attrs`** (*`Dict{Symbol,`*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

`SoftmaxOutput`(...)

Perform a softmax transformation on input, backprop with logloss.

Parameters

- **`data`** (*`SymbolicNode`*) – Input data to softmax.
- **`label`** (*`SymbolicNode`*) – Label data.
- **`grad_scale`** (*`float, optional, default=1`*) – Scale the gradient by a float factor
- **`ignore_label`** (*`float, optional, default=-1`*) – the ignore_label will not work in backward, and this only be used when multi_output=true
- **`multi_output`** (*`boolean, optional, default=False`*) – If set to true, for a (n,k,x_1,..,x_n) dimensional input tensor, softmax will generate n*x_1*. . . *x_n output, each has k classes

- **use_ignore** (*boolean, optional, default=False*) – If set to true, the ignore_label value will not contribute to the backward gradient
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

SwapAxis(...)

Apply swapaxis to input.

Parameters

- **data** (*SymbolicNode*) – Input data to the SwapAxisOp.
- **dim1** (*int (non-negative), optional, default=0*) – the first axis to be swapped.
- **dim2** (*int (non-negative), optional, default=0*) – the second axis to be swapped.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

UpSampling(...)

Perform nearest neighboor/bilinear up sampling to inputs

This function support variable length positional `SymbolicNode` inputs.

Parameters

- **data** (*SymbolicNode[]*) – Array of tensors to upsample
- **scale** (*int (non-negative), required*) – Up sampling scale
- **num_filter** (*int (non-negative), optional, default=0*) – Input filter. Only used by nearest sample_type.
- **sample_type** (*{'bilinear', 'nearest'}, required*) – upsampling method
- **multi_input_mode** (*{'concat', 'sum'},optional, default='concat'*) – How to handle multiple input. concat means concatenate upsampled images along the channel dimension. sum means add all images together, only available for nearest neighbor upsampling.
- **num_args** (*int, required*) – Number of inputs to be upsampled. For nearest neighbor upsampling, this can be 1-N; the size of output will be(scale*h_0,scale*w_0) and all other inputs will be upsampled to thesame size. For bilinear upsampling this must be 2; 1 input and 1 weight.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

abs(...)

Take absolute value of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

ceil(...)

Take ceil value of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

cos(...)

Take cos of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

exp(...)

Take exp of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

floor(...)

Take floor value of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.

- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

log(...)

Take log of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

round(...)

Take round value of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

rsqrt(...)

Take rsqrt of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

sign(...)

Take sign value of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

sin (...)

Take sin of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

sqrt (...)

Take sqrt of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

square (...)

Take square of the src

Parameters

- **src** (*SymbolicNode*) – Source symbolic input to the function
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns

.

14.1.2 Internal APIs

Note: Document and signatures for internal API functions might be incomplete.

_CrossDeviceCopy (...)

Special op to copy data cross device

Parameters

- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.

- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_Div(...)

Perform an elementwise div.

Parameters

- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_DivScalar(...)

Perform an elementwise div.

Parameters

- **array** (*SymbolicNode*) – Input array operand to the operation.
- **scalar** (*float, required*) – scalar value.
- **scalar_on_left** (*boolean, optional, default=False*) – scalar operand is on the left.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_Maximum(...)

Perform an elementwise power.

Parameters

- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_MaximumScalar(...)

Perform an elementwise maximum.

Parameters

- **array** (*SymbolicNode*) – Input array operand to the operation.
- **scalar** (*float, required*) – scalar value.
- **scalar_on_left** (*boolean, optional, default=False*) – scalar operand is on the left.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_Minimum (...)

Perform an elementwise power.

Parameters

- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

_MinimumScalar (...)

Perform an elementwise minimum.

Parameters

- **array** (*SymbolicNode*) – Input array operand to the operation.
- **scalar** (*float, required*) – scalar value.
- **scalar_on_left** (*boolean, optional, default=False*) – scalar operand is on the left.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

_Minus (...)

Perform an elementwise minus.

Parameters

- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

_MinusScalar (...)

Perform an elementwise minus.

Parameters

- **array** (*SymbolicNode*) – Input array operand to the operation.
- **scalar** (*float, required*) – scalar value.
- **scalar_on_left** (*boolean, optional, default=False*) – scalar operand is on the left.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

_Mul (...)

Perform an elementwise mul.

Parameters

- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.

- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_MulScalar(...)

Perform an elementwise mul.

Parameters

- **array** (*SymbolicNode*) – Input array operand to the operation.
- **scalar** (*float, required*) – scalar value.
- **scalar_on_left** (*boolean, optional, default=False*) – scalar operand is on the left.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_NDArray(...)

Stub for implementing an operator implemented in native frontend language with ndarray.

Parameters

- **info** (*, required*) –
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_Native(...)

Stub for implementing an operator implemented in native frontend language.

Parameters

- **info** (*, required*) –
- **need_top_grad** (*boolean, optional, default=True*) – Whether this layer needs out grad for backward. Should be false for loss layers.
- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_Plus(...)

Perform an elementwise plus.

Parameters

- **name** (*Symbol*) – The name of the `SymbolicNode`. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this `SymbolicNode`.

Returns SymbolicNode.

_PlusScalar(...)

Perform an elementwise plus.

Parameters

- **array** (*SymbolicNode*) – Input array operand to the operation.
- **scalar** (*float, required*) – scalar value.
- **scalar_on_left** (*boolean, optional, default=False*) – scalar operand is on the left.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

_Power(...)

Perform an elementwise power.

Parameters

- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

_PowerScalar(...)

Perform an elementwise power.

Parameters

- **array** (*SymbolicNode*) – Input array operand to the operation.
- **scalar** (*float, required*) – scalar value.
- **scalar_on_left** (*boolean, optional, default=False*) – scalar operand is on the left.
- **name** (*Symbol*) – The name of the SymbolicNode. (e.g. *:my_symbol*), optional.
- **AbstractString} attrs** (*Dict{Symbol,*) – The attributes associated with this SymbolicNode.

Returns SymbolicNode.

CHAPTER 15

Neural Networks Factory

Neural network factory provide convenient helper functions to define common neural networks.

MLP (*input*, *spec*)

Construct a multi-layer perceptron. A MLP is a multi-layer neural network with fully connected layers.

Parameters

- **input** (*SymbolicNode*) – the input to the mlp.
- **spec** – the mlp specification, a list of hidden dimensions. For example, `[128, (512, :sigmoid), 10]`. The number in the list indicate the number of hidden units in each layer. A tuple could be used to specify the activation of each layer. Otherwise, the default activation will be used (except for the last layer).
- **hidden_activation** (*Base.Symbol*) – keyword argument, default `:relu`, indicating the default activation for hidden layers. The specification here could be overwritten by layer-wise specification in the `spec` argument. Also activation is not applied to the last, i.e. the prediction layer. See `Activation()` for a list of supported activation types.
- **prefix** – keyword argument, default `gensym()`, used as the prefix to name the constructed layers.

Returns the constructed MLP.

CHAPTER 16

Executor

class Executor

An executor is a realization of a symbolic architecture defined by a `SymbolicNode`. The actual forward and backward computation specified by the network architecture can be carried out with an executor.

bind(*sym*, *ctx*, *args; args_grad=Dict()*, *aux_states=Dict()*, *grad_req=GRAD_WRITE*)

Create an `Executor` by binding a `SymbolicNode` to concrete `NDArray`.

Parameters

- **sym** (*SymbolicNode*) – the network architecture describing the computation graph.
- **ctx** (*Context*) – the context on which the computation should run.
- **args** – either a list of `NDArray` or a dictionary of name-array pairs. Concrete arrays for all the inputs in the network architecture. The inputs typically include network parameters (weights, bias, filters, etc.), data and labels. See `list_arguments()` and `infer_shape()`.
- **args_grad** – TODO
- **aux_states** –
- **grad_req** –

CHAPTER 17

Network Visualization

to_graphviz(*network*)

Parameters

- **network** (*SymbolicNode*) – the network to visualize.
- **title** (*AbstractString*) – keyword argument, default "Network Visualization", the title of the GraphViz graph.
- **input_shapes** – keyword argument, default `nothing`. If provided, will run shape inference and plot with the shape information. Should be either a dictionary of name-shape mapping or an array of shapes.

Returns the graph description in GraphViz `dot` language.

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

Bibliography

[LeNet] Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P., *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, vol.86, no.11, pp.2278-2324, Nov 1998.

[Adam] Diederik Kingma and Jimmy Ba: *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs.LG].

Index